Circus Models for Safety-Critical Java Programs

Frank Zeyda, Lalkhumsanga Lalkhumsanga,
Ana Cavalcanti and Andy Wellings

Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK
Email: frank.zeyda@york.ac.uk,ana.cavalcanti@york.ac.uk,andy.wellings@york.ac.uk

Safety-Critical Java (SCJ) is a restriction of the Real-Time Specification for Java to support the development and certification of safety-critical applications. The SCJ technology specification is the result of an international effort from industry and academia. In this paper, we present a formalisation of the SCJ Level 1 execution model, formalise a translation strategy from SCJ into a refinement notation, and describe a tool that largely automates the generation of the formal models. Our modelling language is part of the *Circus* family; at the core, we have Z, CSP, and Morgan's calculus, but we also use object-oriented and timed constructs from the Oh*Circus* and *Circus* Time variants. Our work is an essential ingredient for the development of refinement-based reasoning techniques for SCJ.

Keywords: Circus; real-time systems; formal models; translation; refinement; RTSJ

1. INTRODUCTION

Java is currently one of the most popular programming languages. Its use in the software industry is extensive. Java, however, has not been widely adopted for development of high-integrity systems, in general, and safety-critical systems, in particular. Concerns about common programming mistakes mean that safer subsets of languages like Ada and C are normally the favoured option. Java, in its full generality, is far too rich a language, and inadequate for time-critical applications due to its heap model based on garbage collection and problems related to prioritisation of threads [1, 2].

As Java implementation technology has matured, the efficiency of the generated code has improved, and new real-time garbage collection algorithms have been developed. As a consequence, some industries have been using Java for their mission-critical applications. Financial trading systems, where real-time performance is critical and applications must be of high-integrity, are, for instance, discussed in [3].

Vendors that support these real-time Java implementations (Atego's Perc Pico product is an example [4]) point out the advantages of using a subset of Java along with highly efficient virtual machines for hard real-time and safety-critical applications. In addition, the supplement DO-332 [5] of the recently released DO-178C certification guidance [6] for the production of airborne software has included more details on how object-oriented programming and virtual machine techniques should be used. This has provided extra encouragement for those interested in using Java in this domain [7, 8].

Most of the proposed approaches to using Java in safety-critical systems build upon real-time extensions of Java. Initially, real-time extensions to Java were *ad hoc*, until the US National Institute of Standards and Technologies brought the communities together to define the requirements for a common standard [9]. As a result, the Real-Time Specification for Java (RTSJ) [10] emerged. This is a version of Java that includes the notion of real-time threads and adopts a region-based memory model [11]. RTSJ has been supported by academia and industry [12, 13, 14], including Oracle (Sun) [15] and IBM [16].

RTSJ, however, is still a very rich language, that encompasses the whole of Java, and includes additional concepts and constructs. This imposes severe challenges in the context of applications that require certification, for instance. As a result, an expert group has been formed to design SCJ, a Java-based language tailored for programming certifiable safety-critical systems.

Safety-Critical Java (SCJ) [17] restricts the Java API, program execution, and memory model in such a way that programs can be effectively analysed for real-time requirements, memory safety, and concurrency issues. This facilitates certification and development of tools that support analysis and verification. SCJ reuses some of RTSJ's concepts and actual API components, but restricts the programming interface. The SCJ technology specification [2] comprises informal descriptions and a reference implementation. Analysis tools have also been developed to establish compliance with the restrictions imposed by SCJ [18].

In recognition of the fact that safety-critical software varies considerably in complexity, there are three compliance levels for SCJ programs and VM

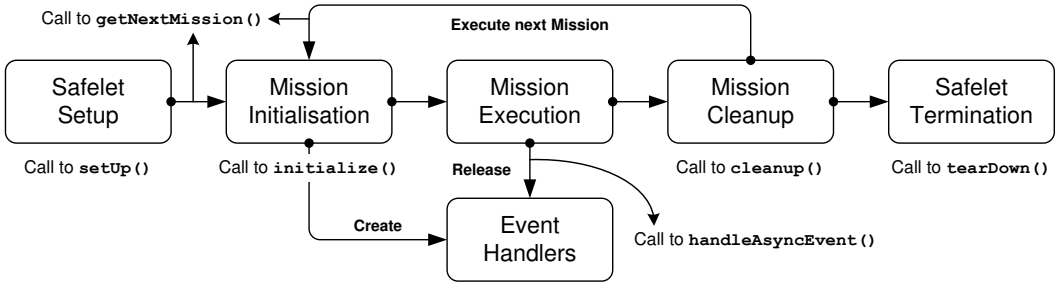

FIGURE 1. Life-cycle of a safelet during execution of a Level 1 application

implementations. In this work, we are concerned with Level 1, which, roughly, corresponds in complexity to the Ravenscar profile for Ada [19]. Level 1 applications support periodic as well as aperiodic event handlers.

The informal account of SCJ [2] relies on text and UML diagrams, and there currently exists no formal account of a semantics for SCJ. The purpose of our work is firstly to define a semantics for SCJ in a language that can be used for refinement (*Circus*). Secondly, we support the automatic generation of formal models of SCJ programs to enable practical use of the semantics for analysis and refinement-based verification. In defining a formal semantics, we clarify subtle aspects of the SCJ programming paradigm, including, for instance, handler interaction and termination. Our formalisation, in particular, targets the mission-based execution model of SCJ.

Circus [20, 21] is a process algebra that integrates well established notations, namely Z [22], CSP [23], and Morgan's refinement calculus [24], to support the development of state-rich reactive systems. Significantly, in *Circus* we can take advantage of mathematical proof to verify concurrent systems that are too large for model checking. *Circus* is currently being used to verify software in aerospace applications, including software for the Chinese manned lunar lander, and novel virtualisation software by the US Naval Research Laboratory. *Circus* has a formal semantics [25], and a refinement theory and strategy [21]. A specialised *Circus*-based refinement technique permits the analysis of control systems specified in Simulink [26, 27].

The semantics of *Circus* is based on the Unifying Theories of Programming (UTP) [28]. The UTP supports well the combination of notations like Z and CSP, and also allows us to consider constructs from other paradigms. Extensions and variants of *Circus* cover, for instance, aspects of time and mobility. We use its object-oriented variant, *OhCircus* [29], as our base notation. *OhCircus* can be used to model both data objects and the active behaviour of the SCJ components. The UTP-based semantics of *OhCircus* enables us to combine *OhCircus* with *Circus* Time, a version of *Circus* with support for timed behaviours.

Our work firstly elicits the conceptual behaviour of the SCJ framework, and secondly formalises how the translation of actual SCJ programs into their *OhCircus* specifications can be achieved in a traceable manner. We ignore certain aspects of SCJ, such as the memory model, which we discuss in a separate paper [30], and scheduling policy. Our focus is the top-level design and execution of SCJ programs, and SCJ's primary framework and application components.

In our view, the SCJ framework as designed in Java embeds a general programming paradigm. SCJ adopts a particular approach to data operations, memory management, and event-based versus thread-based program designs [31]. The fact that it can be realised on top of Java and the RTSJ is a bonus. It is conceivable to implement specific support based on other mainstream languages, or even define an entirely new language. Our model identifies the fundamental concepts of SCJ at a level at which it can be itself regarded as a language.

Regarding imperative constructs, there are a number of features of the Java language that reduce clarity or are otherwise challenging to describe and analyse formally. An additional contribution of our work is to identify and exclude such constructs, which are often similar to those prohibited by subsets of C and Ada [32] used in the safety industry. We note that the design of SCJ [2] does not address constraints on statements. Although we do not claim here to identify all constructs (the ones that have simple models in *Circus* we admit), our work can nevertheless be seen as a first step in defining a safe subset for SCJ.

We formalise the translation from safety-critical Java into *Circus* using a collection of compositional translation rules. To automate the translation process, we present an annotation framework and a tool that can generate models for arbitrary SCJ programs that satisfy our restrictions and are suitably annotated.

Our work shows that *Circus* is adequate for capturing faithfully the semantics of SCJ programs respecting our restrictions, and that the construction of *Circus* models can be automated. This is provided the program code has been annotated, which requires human interaction. No in-depth knowledge of *Circus* is required to drive the model generation process and the models we produce can act as the targets for a refinement strategy whose application provides opportunities for automation, too.

We have validated our *Circus* models in FDR [33]

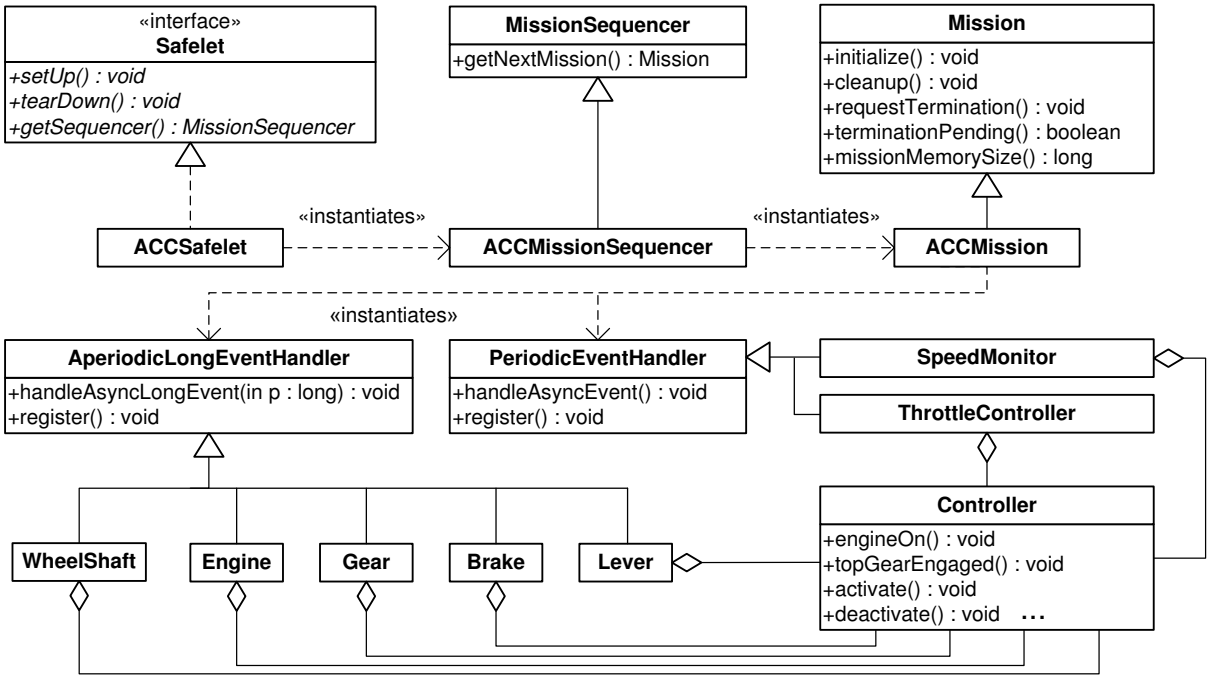

FIGURE 2. UML class diagram for the cruise controller

after translating them into CSP. In doing so, we managed to show livelock and deadlock freedom of simple applications. We also used the CZT parser and type-checker [34] to ensure that the *Circus* specifications generated by our tool are valid, and the tool itself validates the feasibility of our automatic approach. Recent work [35, 36] reports on a refinement strategy that transforms abstract, centralised *Circus* specifications that are structured in terms of behavioural and timing requirements into concrete models that adhere precisely to the structure of models presented here.

In summary, what we describe in this paper is first a precise semantics for core elements of SCJ. This enables formal verification of SCJ applications beyond the informal validation of statically-checkable properties currently available [18]. *OhCircus* provides a notion for refinement, and our work is an essential step to justify future development and verification methods that can produce high-quality SCJ implementations. For verification, we can construct models of particular programs, and use the *Circus* and UTP techniques for reasoning. For development, we can start from an abstract specification, and develop implementations that respect the restrictions of our models [36]. The latter approach can also be used for verification of a given program with respect to a specification. For that, we need to guide the refinement strategy to produce a *Circus* model that is syntactically equivalent to the model of the SCJ program, as it is generated by our technique and tool. Preliminary results are reported in [35], and we note that this approach is more flexible than mere code generation as it enables the verification of arbitrary programs, and, in particular, supports hand-coded optimisations.

In comparison to our previous work [37], here we elaborate and make precise our modelling approach and formalise its construction. We also modularise and, in several places, simplify the framework model, which captures the generic behaviour of the SCJ paradigm.

The structure of the article is as follows. Section 2 discusses preliminary material: SCJ, a cruise controller, which is used as a running example, and the *Circus* family of languages. The next three sections discuss in detail the *Circus* model of SCJ: Section 3 explains the top-level architecture; Section 4 presents the fixed framework model; and Section 5 presents the program-specific application model. In Section 6 we present a formalised translation strategy from SCJ into *Circus*, and Section 7 examines issues related to automation and tool support. Lastly, in Section 8 we conclude and address related and future work.

2. PRELIMINARIES

Here, we give an overview of the SCJ execution model and introduce an example: an automotive cruise controller. Afterwards, we present *Circus* and *OhCircus*, the formal notations in which our models are written.

2.1. Safety-Critical Java

The SCJ programming paradigm is based on the notion of missions. They are sequentially executed by an application-defined mission sequencer provided by a safelet, the top-level entity of an SCJ application. The life-cycle of a safelet is illustrated in Fig. 1. Conceptual

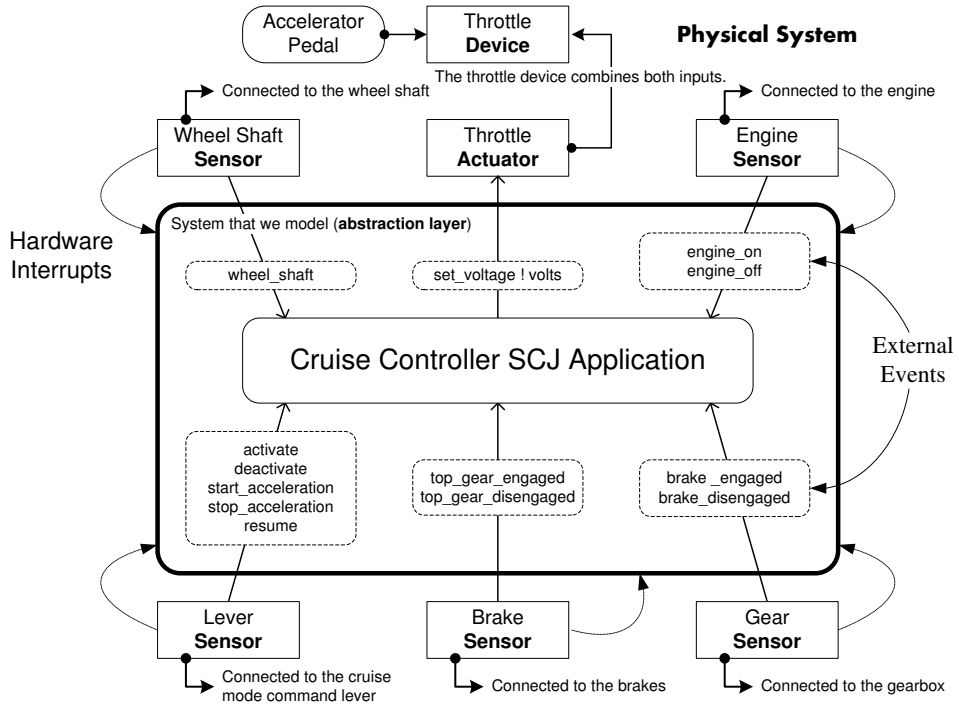

FIGURE 3. ACC system interactions.

entities are realised by either interfaces or abstract classes. Namely, they are the `Safelet` interface, and the abstract classes `MissionSequencer` and `Mission` (see Fig. 2 for a UML diagram).

As already mentioned, in this work, we consider only SCJ Level 1 applications. A Level 1 mission executes in parallel a set of asynchronous event handlers (both periodic and aperiodic handlers are supported). Each aperiodic handler is associated with a set of events: firing one of them causes a handler method to be scheduled for execution. Periodic event handlers, on the other hand, are controlled by a timer. Event handlers are defined by extending abstract classes whose handling method must be implemented by a concrete subclass. These classes are, in particular, `PeriodicEventHandler`, `AperiodicEventHandler` and `AperiodicLongEventHandler` (see Fig. 2).

2.2. A cruise control system

As an example of an SCJ Level 1 program, and to illustrate our modelling approach, we present an implementation of the automotive cruise control system (ACC) in [10]. The example was first published as a case study in [38]. It therefore does not reflect the current state-of-the-art in automotive technology, but is nevertheless sufficient for illustrating our method.

The purpose of an ACC is to maintain the speed of a vehicle to a value set by the driver. In Fig. 3 we give an overview of its main components and commands. Explicit commands are given by a lever whose positioning corresponds to the following instructions: *activate*, to turn on the ACC if the car is in top gear, and maintain (and remember) the current speed; *deactivate*, to turn off the ACC; *start_accelerating*, to accelerate at a comfortable rate; *stop_accelerating*, to stop accelerating and maintain (and remember) the current speed; and *resume* to return to the last remembered speed and maintain it. When the driver operates the brake pedal, changes gear, or switches off the engine, the ACC is deactivated. Besides, when the engine is switched on, the ACC is initialised in such a way that the *resume* command cannot be issued as no speed is initially recorded.

The speed of the vehicle is measured via the rotations of a shaft that is connected to one of the wheels. The speed of the car is controlled by the ACC using the throttle position, which is determined by the depression of the accelerator pedal and a voltage supplied by the ACC. The combination of these values is performed by a mechanism outside the ACC.

Sensors detect external events and generate appropriate interrupts, as shown in the diagram in Fig. 3. Their service routines in the SCJ program determine the precise interrupting event and fire a corresponding SCJ event that releases one of the aperiodic handlers. For the setting of the throttle voltage, communication of the new voltage value to the throttle actuator is realised in the program using a hardware data register.

Fig. 2 presents a UML class diagram that gives an overview of the design of the ACC as an SCJ Level 1 safelet. In Fig. 1 we also highlight the sequence of method calls that are carried out by the SCJ run-time environment (virtual machine) to execute a safelet. We next discuss the entities of the ACC individually.

```
public void initialize() {
  /* Create SCJ events and interrupt handlers. */
  createEvents();
  createISRs();
  registerISRs();
  /* Create event handlers and shared data. */
  WheelShaft shaft = new WheelShaft(shaft_event);
  SpeedMonitor speedo = new SpeedMonitor(shaft, 500);
  ThrottleController throttle =
    new ThrottleController(speedo);
  Controller cruise = new Controller(throttle, speedo);
  Engine engine = new Engine(cruise, engine_event);
  Brake brake = new Brake(cruise, brake_event);
  Gear gear = new Gear(cruise, gear_event);
  Lever lever = new Lever(cruise, lever_event);
  /* Register event handlers with the mission. */
  shaft.register();
  engine.register();
  /* ... */
}
```

FIGURE 4. Mission initialisation method.

Safelet. `ACCSafelet` is the entry point for the SCJ application. This class provides the method `getSequencer()` that returns the application's mission sequencer. The other two methods `setUp()` and `tearDown()` are provided for initialisation and cleanup tasks.

Mission Sequencer. The `ACCMissionSequencer` class constructs instances of the `Mission` class, by implementing `getNextMission()`. These instances determine the missions sequentially executed by the safelet. We note that in the ACC there is only one mission.

Missions. Concrete subclasses of `Mission` have to implement the `initialize()` and `missionMemorySize()` methods. The former creates and registers the periodic and aperiodic event handlers of the mission. Fig. 4 includes the definition of the `initialize()` method in the ACC implementation. We first have calls to methods that create SCJ events and interrupt service routines (ISRs). The SCJ events are held by instance variables of the class and we omit a detailed discussion of the ISRs. The subsequent statements create the periodic and aperiodic event handlers of the application, as well as the shared `Controller` object. Finally, a cascade of calls to `register()` on the handler objects registers them with the current mission.

The two extra methods `requestTermination()` and `terminationPending()` of the mission class are `final` and so cannot be overridden. They allow for the mission to be terminated by one of the handlers. As part of the termination process, the `cleanup()` method is called by the SCJ infrastructure to enable the execution of application code for mission-specific cleanup tasks.

Handlers. Periodic event handlers implement the method `handleAsyncEvent()`, and aperiodic event handlers implement either `handleAsyncEvent()` or `handleAsyncLongEvent(int)` to specify their behaviour when the handler is released, depending on whether the handler class is derived from `AperiodicEventHandler` or `AperiodicLongEventHandler`. The difference between the latter two is that `AperiodicLongEventHandler` supports the passing of a `long` parameter to the handler method, whereas in `AperiodicEventHandler` the handler method is parameterless.

In the ACC, we use `AperiodicLongEventHandler` for all aperiodic handlers, except only for `WheelShaft`, and use the parameter to identify the external event that caused the release of the handler. Based on its value, the handler method selects a method to call on the shared `Controller` object. The passing of the value identifying the event to the handler is realised by an interaction mechanism that is implemented at the level of interrupt service routines (ISRs). The mechanism queries the particular sensor that caused the interrupt and releases the corresponding handler while passing the corresponding event identifier.

Beyond the ACC example. Fig. 2 does not show all components of the SCJ API. There are eight classes that realise the mission framework, twelve classes in the handler hierarchy, five classes that deal with real-time threads, seven classes concerned with scheduling, and ten classes for the memory model. The formal model that we present here abstracts from all these details of the realisation of the SCJ Level 1 programming paradigm in Java. We capture the main concepts of this novel execution model. This enables reasoning based on the core components of the SCJ paradigm.

The memory model of SCJ employs the region-based approach of RTSJ. Since SCJ does not support garbage collection, (a restricted version of) scoped memory is permitted, but not heap memory. Scoped memory has a limited life-span that is controlled by the SCJ run-time environment. Memory scopes form a tree, and restrictions on references between scopes effectively alleviate the problem of dangling references.

At the root of the tree we have immortal memory. It contains objects that are never deallocated during the execution of the safelet. Below, mission memory is created and exists for the duration of executing a mission, and is used to store objects that are shared between handlers. Further below, each handler executes in its own per release memory area whose life-time is limited to the execution of the handler's `handleAsyncEvent()` method. Additionally, handlers may create their own private memory areas as needed.

Before considering the formalisation of SCJ programs, we present our modelling notation next.

2.3. The *Circus* family of languages

Like in CSP, the key elements of *Circus* models are processes that interact with each other and their environment via communication channels. Unlike CSP, *Circus* processes may encapsulate a state. This renders *Circus* useful for both model-based and behavioural specification [39]. In what follows, we introduce standard *Circus* and its derivatives *OhCircus* and *Circus* Time. We recall that our modelling notation is indeed a combination of these three languages.

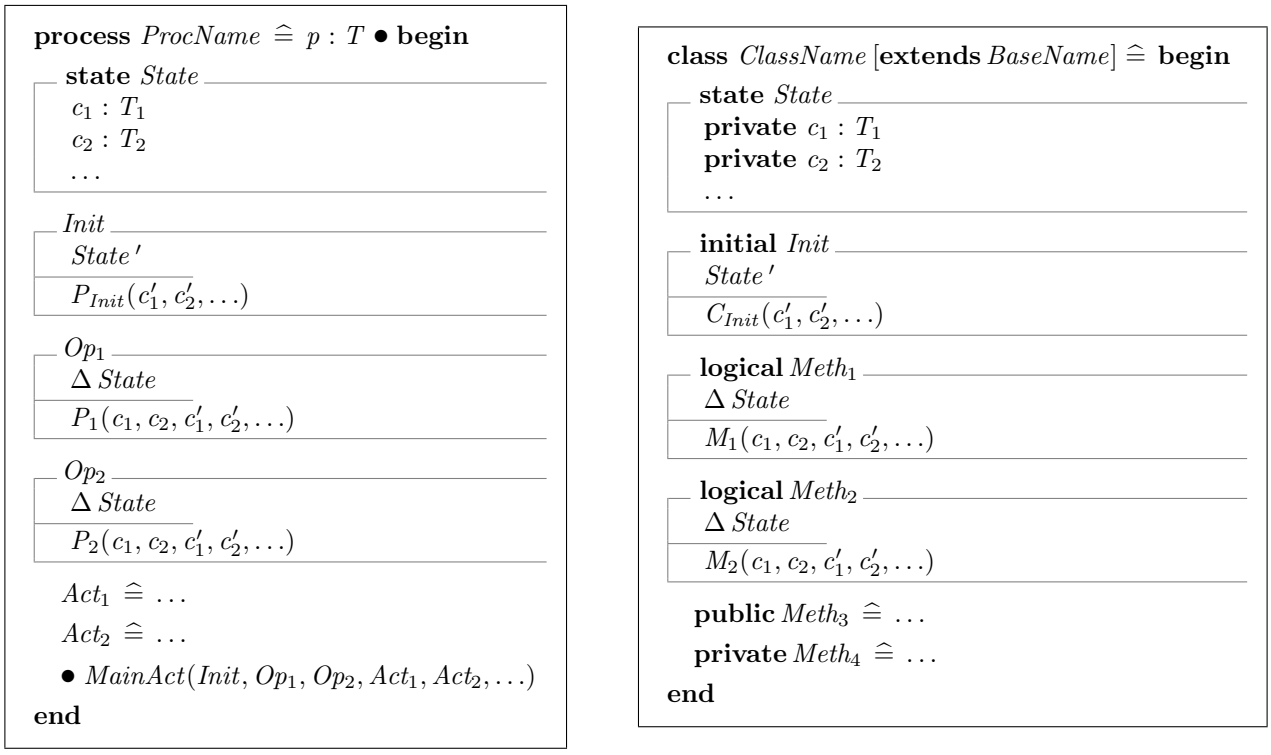

FIGURE 5. Typical examples of a *Circus* process (left) and an *OhCircus* class (right).

2.3.1. Circus

The typical syntax of a *Circus* process is sketched in the left diagram of Fig. 5. The name of the process is *ProcName* and p is a parameter of type T (in a parameterless process, p is absent). The process state is determined by the Z schema in the **state** paragraph. This schema is called *State* here and has the components c_1, c_2, and so on, of types T_1, T_2, and so on. Because the state is local to the process, it is only visible by the local schema operations and actions. Here, they are *Init*, Op_1, Op_2, Act_1, Act_2, and so on, as well as the main action *MainAct* at the end after the '•'.

A process usually contains an initialisation action (*Init* above). It is typically specified by a Z operation schema whose predicate constrains the values of dashed variables only. Dashed names, as common in Z, are used to refer to the value of variables after execution of some operation. Dashing a schema, as in the declaration *State'* in *Init*, renames the components of that schema to their dashed counterparts.

Actions can be defined using Z data operation schemas, constructs from CSP, as well as commands from Morgan's calculus to modify the state. Data operations typically change the state, and their specification constrains the values of the state components c_1, c_2, and so on, and of the corresponding variables c_1', c_2', and so on. The former refer to the values of the state components before the operation, and the latter, as mentioned above, represent the values of the state components after the operation. All these variables are implicitly declared by $\Delta\, State$. We note that data operations are always atomic in *Circus*.

The main action (*MainAct* in Fig. 5) defines the behaviour of the process. It may reference local actions, which are introduced primarily for structuring purposes. Similarly, local actions may reference other local actions in their bodies, too.

The language of actions provides a rich set of constructs. Table 1 includes all action constructs inherited from CSP that are relevant for the models in this article. As a notational convention, A stands for an action, c for a channel, x for a variable, T for a type, cs for a set of channels, ns for a set of variables, and E for an expression yielding a value. We provide a brief explanation of each action operator in the sequel, but postpone a more detailed discussion to where the operators are first used in our models.

We first have the **skip** action, which terminates immediately without changing the state. Next, the action **abort** is the bottom of the refinement lattice for actions and thus represents (program) failure. Prefixes are used for communication with the environment or other processes. They can take the form of simple synchronisations ($c \longrightarrow A$), inputs ($c\,?\,x \longrightarrow A(x)$), or outputs ($c\,!\,E \longrightarrow A$).

Actions may be combined in various ways. External choice ($A_1 \,\Box\, A_2$) is a choice that is resolved by the environment. This means that the first interaction determines which of the two actions is chosen, and the communications offered are the ones offered by either A_1 or A_2. If the same communication is offered by both actions, the choice becomes nondeterministic. Sequential composition ($A_1 \,;\, A_2$) has its usual meaning.

Parallelism ($A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2$) executes

Action	Syntax	Description
Skip	**skip**	Immediately terminates without changing the state.
Diverge	**abort**	Divergent action: it may not terminate and represents failure.
Synchronisation	$c \longrightarrow A$	Simplest form of an interaction: no value is communicated.
Input Prefix	$c\,?\,x \longrightarrow A(x)$	Binds the variable x to the value read through the channel c.
Output Prefix	$c\,!\,E \longrightarrow A$	Outputs the value of the expression E on the channel c.
External Choice	$A_1 \,\Box\, A_2$	A choice that is resolved by the environment.
Sequence	$A_1 \,;\, A_2$	Executes the two actions A_1 and A_2 in sequence.
Parallelism	$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2$	Parallelism in which actions have to synchronise on the channels in cs.
Interleaving	$A_1 \,[\![\, ns_1 \mid ns_2 \,]\!]\, A_2$	Parallelism where no synchronisation between A_1 and A_2 is required.
Iterated Interleaving	$\|\|\| x : S \bullet A(x)$	Interleaving of all actions $A(x)$ where $x \in S$.
Interrupt	$A_1 \,\triangle\, c \longrightarrow A_2$	A synchronisation on c interrupts the execution of A_1 and subsequently transfers control to A_2.
Hiding	$A \setminus cs$	Interactions via channels in cs are hidden and take place immediately when they are enabled.
Recursion	$\mu X \bullet F(X)$	Occurrences of X in F constitute recursive calls.

TABLE 1. *Circus* action constructs derived from CSP.

two actions concurrently. That is, they may progress independently but have to synchronise on the channels in the synchronisation set cs. Importantly, parallel actions must write to disjoint parts of the process state to avoid write conflicts. To achieve this, the name sets ns_1 and ns_2 determine the state components that each parallel action is allowed to modify. For the operator to be well formed, the name sets hence have to be disjoint. Interleaving of actions, written as $A_1 \,[\![\, ns_1 \mid ns_2 \,]\!]\, A_2$, is a special case of parallelism where the set of synchronisation channels is empty. We also have an iterated interleaving that interleaves all actions $A(x)$ where x ranges over the elements of some set S.

There are two noteworthy points about *Circus* parallelism and interleaving. Firstly, both operators terminate only when all parallel actions have terminated. Secondly, they enforce non-interference: state changes become visible only after termination of the operator. This is important to preserve monotonicity.

An interrupt $A_1 \,\triangle\, c \longrightarrow A_2$ is used to transfer control from one action A_1 to another action A_2 at any point during the execution of A_1. This happens as a consequence of a synchronisation on the channel c. In $A \setminus cs$, the channels in the set cs are hidden in the execution of the action A. Synchronisations on hidden channels take place internally and as soon as they are enabled, without visibility or participation of the environment. Lastly, the recursion construct $\mu X \bullet F(X)$ names the action $F(X)$ in its body X, and so uses of X in $F(X)$ are recursive calls.

State operations can be specified either by Z operation schemas such as Op_1 and Op_2 in Fig. 5, or guarded commands. A list of all guarded commands used by our models is given Table 2. Assignments change the value of a state component or local variable. Local variables are introduced by way of a local variable block of the form $\mathbf{var}\, x : T \bullet A(x)$, where x becomes a local variable of type T that A may refer to.

The conditional statement is in Dijkstra's generalised form [40], taking a list of guarded commands (the g_i are predicates). A binary conditional $\mathbf{if}\ b\ \mathbf{then}\ A_1\ \mathbf{else}\ A_2$ is hence written $\mathbf{if}\ b \longrightarrow A_1 \,[]\, \neg\, b \longrightarrow A_2\ \mathbf{fi}$. Lastly, a value parameter p of type T can be introduced using $\mathbf{val}\, p : T \bullet A(p)$. This is useful in order to define parametrised actions.

Processes can be defined explicitly in the form sketched in Fig. 5, or alternatively, similar to actions, by virtue of CSP operators over existing process definitions, like those presented in Table 1. The only difference in comparison to the action operators is that parallel composition and interleaving do not require name sets, since processes fully encapsulate their state.

2.3.2. OhCircus

OhCircus [29] extends *Circus* with an additional notion of class. Unlike processes, classes define data objects, and such objects can be used in arbitrary mathematical expressions. In contrast, processes are not values and, because of that, can only be used as static entities of a model. A semantics to cater for the use of processes as values introduces considerable complications, in particular, in dealing with inheritance. The distinction that is made between processes and classes results in more tractable models in terms of available reasoning techniques. In general, processes, with their interacting capabilities, describe the active behaviour of a model. Classes model passive data objects, and operations on these objects are defined by methods of the class.

The typical form of an *OhCircus* class definition is sketched in the right diagram of Fig. 5. The access modifiers **public**, **private** and **protected** are included, but have no semantic significance apart from controlling visibility. The modifier **logical** expresses the intention of a logical (specification) method, which does not have to be retained during refinement.

The permissible notation for *OhCircus* class methods includes all schema operations, guarded commands, and some additional object-oriented constructs used

Action	Syntax	Description
Assignment	$x := E$	Changes the value of a state component or local variable.
Local Variable	$\mathbf{var}\, x : T \bullet A(x)$	Declaration of a local variable x.
Conditional	$\mathbf{if}\, g_1 \longrightarrow A_1 \,[]\, g_2 \longrightarrow A_2 \,[]\, \ldots \,[]\, g_n \longrightarrow A_n\, \mathbf{fi}$	Executes an action whose guard g_i is true.
Value Parameter	$\mathbf{val}\, p : T \bullet A(p)$	Action with a value-parameter p of type T.

TABLE 2. *Circus* action constructs used from Morgan's calculus.

Construct	Syntax	Description
Current Object	**this**	References the current object.
Superclass Object	**super**	References the superclass object.
Selection	$obj\,.\,[\texttt{field}\,/\,\texttt{method}]$	Accesses a field or method of an object obj.
Object Creation	**new** $Class(args)$	Creates a new object of class type $Class$.
Synchronised Method	**sync** $Meth(args) \mathrel{\widehat=} \ldots$	Declares a method $Meth$ as atomic.
Simple Delay	**wait** t	Waits for t time units.
Nondeterministic Delay	**wait** $t_1 \,..\, t_2$	Waits between t_1 and t_2 time units.

TABLE 3. Relevant *OhCircus* and *Circus* Time constructs.

to instantiate new data objects, invoke methods, access object fields, and support inheritance. Table 3 includes all *OhCircus* constructs that are relevant for the material in this article. We have the **new** construct to create a new class object, and the **this** and **super** keywords to refer to fields and methods of the current and superclass. In summary, the notation for methods is similar to the notation for actions, but lacks the CSP operators. Besides, methods can be declared as synchronised using the modifier **sync**. Synchronised methods are an extension we introduce to the *OhCircus* language. This enables us to treat method execution as atomic, just like Z data operations.

2.3.3. Circus Time

Circus Time [41] is an extension of *Circus* to model timed behaviours. Subsequently, we make use of two further operators of *Circus* Time. They are the **wait** t and **wait** $t_1 \,..\, t_2$ statements, also included in Table 3. The first delays execution by t time units, and the second is a nondeterministic delay that may wait between t_1 and t_2 time units. Apart from the use of *Circus* Time, object references from our previous SCJ memory model in [30] are also used. They are specified at the level of the Unifying Theories of Programming [28], the common semantic framework of *Circus* and its extensions.

In the next three sections, we present our model for SCJ programs. We have given here a brief overview of the main features of the *Circus* family of languages used in formulating those models. Extra details of the notation are explained as we discuss the models.

3. MODEL ARCHITECTURE

Our models factor into two dimensions: a generic framework model, and an application model that corresponds to a particular concrete SCJ program. In the framework model, we specify the semantics of the safelet, mission sequencer, missions, and event handlers. They are the fundamental building blocks of Level 1 applications. To illustrate the architecture of application models, we make use of the cruise controller example presented in the previous section.

Fig. 6 presents an overview of the structure of the complete model of a typical SCJ Level 1 application — here the cruise controller. Each box represents a *Circus* process and is labelled by the process name. Boxes inside the large surrounding rectangle denote a process that belongs to the framework model. These processes capture the generic behaviour of the SCJ programming paradigm. Boxes outside (highlighted in grey) denote processes that belong to the application model. These are in direct correspondence with the classes of an SCJ program. While framework processes (suffix FW) have fixed names, application processes (suffix App) carry the names of their respective Java classes. The three dots indicate that a few of the event handler framework and application processes of our case study are omitted. They integrate into the model in a similar fashion as the handler processes shown in Fig. 6.

The *Circus* model of the entire SCJ application is obtained by parallel composition of the framework and application models. The framework model is itself a parallel composition, namely of all framework processes. Similarly, the application model is defined by a parallelism involving all application processes.

Arrows in Fig. 6 indicate the channels on which the processes communicate (synchronise). The framework processes *SafeletFW* and *MissionSequencerFW*, for instance, synchronise on *start_sequencer* and *done_sequencer*. These two channels control the execution of the mission sequencer.

We distinguish between control channels and method channels; the latter are identified by the suffixes *Call* and *Ret*. To give an example, *start_mission* is a control channel whereas *requestTerminationCall* and *requestTerminationRet* are method channels. All calls to infrastructure methods are modelled by channel

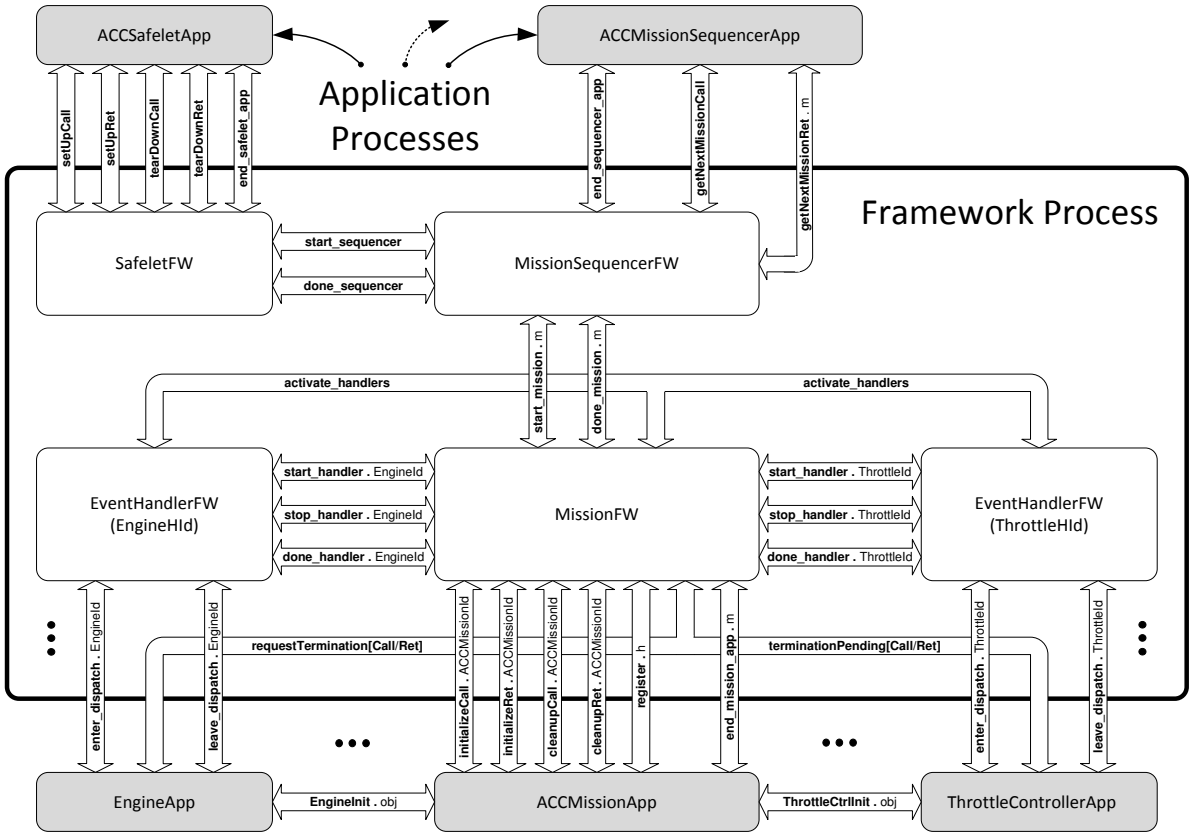

FIGURE 6. Structure of the model for the SCJ cruise controller.

communications to enable the framework processes to trigger or respond to calls to those methods. In the application processes, a call to `requestTermination()`, for instance, has to interact with *MissionFW*, the mission framework process. Some of the channel communications are parametrised by a value (notation $c\,.\,x$). In Fig. 6 we use m and h as placeholders for arbitrary mission and handler identifiers.

In the framework model, we have exactly one process to model execution of the safelet (*SafeletFW*), the mission sequencer (*MissionSequencerFW*) and the missions (*MissionFW*). In contrast, we have an instance of the *EventHandlerFW* framework process for each handler. This process is parametrised by an identifier of a given type *HandlerId*. We introduce one such identifier for each handler object in the program.

Whereas the *FW* processes are defined once and for all, we require a translation technique to construct the *App* processes for particular SCJ applications. Translation entails the definition of channels, channel sets, processes, and *OhCircus* classes to model the SCJ application classes. We note that Fig. 6 does not include a process for all classes of the cruise controller application, only those that implement or extend an SCJ infrastructure class. For this reason, we have no process for the `Controller` class which, acting as a data object, only has a model in terms of an *OhCircus* class.

To elaborate our account of the architecture further, Fig. 7 presents a more application-centred view of the *Circus* model. Here, the framework model is characterised by a single process *Framework*, which corresponds to the large enclosing box in Fig. 6. Synchronisations on method channels are, for brevity, subsumed into single arrows. Fig. 7 also illustrates some of the data objects. For the mission sequencer, the mission and each handler, we have an *OhCircus* class (suffix *Class*) that encodes an underlying data object, and a *ControllerClass*, which models the `Controller` class. All shared data objects are created by *ACCMissionApp* during initialisation of the mission. We omit some of the event handler classes for brevity.

At the system level, all method and control channels are hidden. The only observable interactions are the ones with the external environment. We can see in Fig. 7 that, specifically, the application processes for handlers engage in these interactions. The external channels are application-specific and define the interface of the system. For example, we declare basic channels *engine_on* and *engine_off* in the cruise controller model to represent the events that occur when the engine is switched on and off. Table A.1 in Appendix A includes all channels that are used for external events of the cruise controller. The free type *LEVER* is introduced to represent the five positions of the command lever. In the program, these interactions correspond to device accesses and hardware interrupts (see Fig. 3).

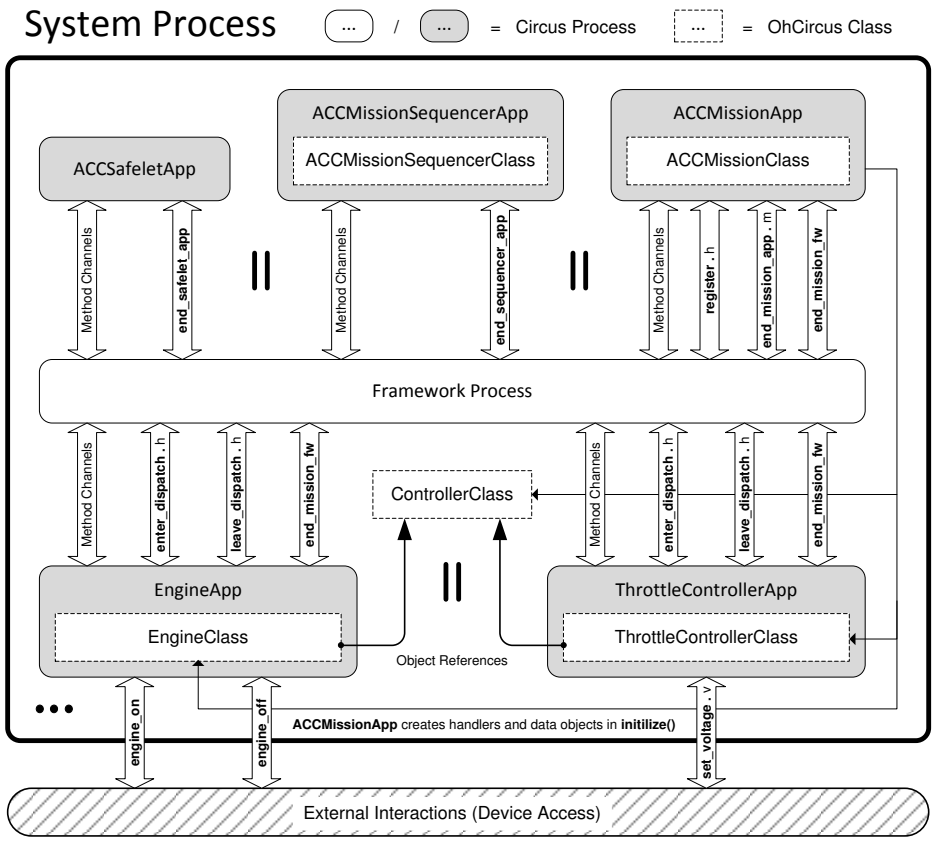

FIGURE 7. System view of the SCJ cruise controller model.

In what follows, we describe each of the processes of our model in more detail.

3.1. System Process

The *System* process models the entire application. Its uniform definition is sketched below.

$$\textbf{process}\ System \mathrel{\widehat=} \left(\begin{array}{l} Framework \\ \quad [\![\ SCJMethChan \\ \quad\ \cup \\ \quad\ AppControlChan\ ]\!] \\ Application \end{array}\right) \setminus \left(\begin{array}{l} SCJMethChan \\ \cup \\ AppControlChan \end{array}\right)$$

Above, *Framework* and *Application* refer to the individual composite processes that model the framework and application classes. Two channel sets are used to define the synchronisation set of the parallelism. They are *SCJMethChan* for all method channels and *AppControlChan* for control events used by the framework to control the application processes. Internal control channels of the framework are already hidden in the *Framework* process. We hide all method and application control channels so that only the channels that constitute the external interface are exposed.

To define the *Application* process, we first specify a process *HandlersApp*, which combines all handler application processes in parallel. For the ACC model, this process is defined as shown below.

$$\textbf{process}\ HandlersApp \mathrel{\widehat=} \left(\begin{array}{l} WheelShaftApp\ [\![\ \{\!|\ end\_mission\_fw\ |\!\}\ ]\!] \\ EngineApp\ [\![\ \{\!|\ end\_mission\_fw\ |\!\}\ ]\!] \\ BrakeApp\ [\![\ \{\!|\ end\_mission\_fw\ |\!\}\ ]\!] \\ GearApp\ [\![\ \{\!|\ end\_mission\_fw\ |\!\}\ ]\!] \\ LeverApp\ [\![\ \{\!|\ end\_mission\_fw\ |\!\}\ ]\!] \\ SpeedMonitorApp\ [\![\ \{\!|\ end\_mission\_fw\ |\!\}\ ]\!] \\ ThrottleControllerApp \end{array}\right)$$

The only synchronisation required is on the channel *end_mission_fw* which, beyond other purposes discussed later on, cumulatively terminates the handler application processes when the program terminates.

With the above, the *Application* process is defined by way of an interleaving (operator $|||$) as follows.

$$\textbf{process}\ Application \mathrel{\widehat=} \left(\begin{array}{l} ACCSafeletApp \\ \quad ||| \\ ACCMissionSequencerApp \\ \quad ||| \\ ACCMissionApp \\ \quad ||| \\ HandlersApp \end{array}\right)$$

In defining *Application* for the cruise controller, we account for all grey boxes in Fig. 7. The definition

process $SafeletFW \mathrel{\widehat=}$ **begin**
$SetUp \mathrel{\widehat=} setUpCall \longrightarrow setUpRet \longrightarrow \mathbf{skip}$
$Execute \mathrel{\widehat=} start\_sequencer \longrightarrow done\_sequencer \longrightarrow \mathbf{skip}$
$TearDown \mathrel{\widehat=} tearDownCall \longrightarrow tearDownRet \longrightarrow \mathbf{skip}$
$\bullet\ SetUp\ ;\ Execute\ ;\ TearDown\ ;\ end\_safelet\_app \longrightarrow \mathbf{skip}$
end

FIGURE 8. Framework process for the safelet.

reflects the four main entities in the mission model: the safelet, mission sequencer, missions and handlers. In applications where there is more than one mission, we construct a process *MissionsApp* for them just like *HandlersApp*. The use of interleaving in the application model highlights that, although application processes individually synchronise with the framework, they do not synchronise with each other. The exception to this are handlers that are released by the same external event or handlers that release other handlers. In the cruise controller, however, there are no such handlers.

Having presented the top-level model view, in the next two sections we elaborate on the detailed specification of each framework (Section 4) and application (Section 5) process.

4. FRAMEWORK MODEL

In this section, we present the framework model for the safelet, mission sequencer, missions, and handlers. We recall that the *Circus* processes in this model are fixed: they are the same for every SCJ application and embody the generic behaviour of the SCJ paradigm.

4.1. Safelet

The framework process *SafeletFW* is given in Fig. 8. It has no state. The main action sequentially executes the *SetUp*, *Execute* and *TearDown* local actions. They initiate and wait for completion of the initialisation, execution and cleanup phases of the safelet. This is followed by a synchronisation on *end_safelet_app* to terminate the safelet application process, after which the safelet framework process itself terminates.

The actions *SetUp* and *TearDown* synchronise in sequence (prefix operator $\longrightarrow$) on the $setUp[Call/Ret]$ and $tearDown[Call/Ret]$ channels, before terminating. The synchronisations model calls to the methods `setUp()` and `tearDown()` of the SCJ application class that implements the `Safelet` interface. To illustrate the calling mechanism, we recall that in the system model (Fig. 6), all framework and application processes are composed in parallel, and in that parallel composition, *SafeletFW* and the respective safelet application process (*ACCSafeletApp* in Fig. 6) synchronise on the aforementioned method channels. Hence, we obtain a behaviour that can be described as sketched by the parallel action fragment below, though the actions are embedded in processes.

$$(setUpCall \longrightarrow \mathbf{skip}\ ;\ setUpRet \longrightarrow \mathbf{skip})$$
$$[\![\varnothing \mid \{\!|\, setUpCall, setUpRet, \ldots \,|\!\} \mid \ldots]\!]$$
$$\left(\begin{array}{l} \ldots\ ; \\ setUpCall \longrightarrow MethBody\ ;\ setUpRet \longrightarrow \mathbf{skip}\ ;\ \ldots \end{array}\right)$$

The left action of the parallelism comes from the *SetUp* action of the *SafeletFW* process, and the right action is in the safelet application process. The parallel actions first synchronise on the channel *setUpCall*, after which the right parallel action executes the action *MethBody* of the method body of the `setUp()` method. Only then can the left action make further progress, resulting in a synchronisation on *setUpRet* with the left-hand parallel action terminating.

Since the methods of the safelet are parameterless and do not return any values, the communications are synchronisations: there is no input or output. As noted, the methods themselves are specified in the application process for the safelet, which is discussed in the next section. As a convention, we use capitalised names for actions and lower-case names for channels. In cases where the names of model entities are derived directly from identifiers in the program, this does not apply.

Execute also performs synchronisations, but instead raises two framework events: *start_sequencer* to start the mission sequencer, and *done_sequencer* to wait for its termination. The framework process for the mission sequencer component is specified next.

4.2. Mission Sequencer

The mission sequencer process communicates with the safelet process to determine when it has to start, and also to signal its termination. Its specification in *Circus* is presented in Fig. 9. The main action executes *Start*, which waits for the mission sequencer to be started, as signalled by a synchronisation on *start_sequencer*. Afterwards, execution proceeds as specified by the recursion in the action *Execute*. In each iteration, we synchronise on the channels *getNextMissionCall* and *getNextMissionRet* to obtain the next mission *next*. This corresponds to a call to the SCJ method `getNextMission()` of the concrete `MissionSequencer` application class. Since this call returns a (mission) object, *getNextMissionRet* receives as an input a value

process *MissionSequencerFW* $\widehat{=}$ **begin**
  Start $\widehat{=}$ *start_sequencer* $\longrightarrow$ **skip**
  Execute $\widehat{=}$ $\mu X \bullet$ *getNextMissionCall* $\longrightarrow$ *getNextMissionRet* ? *next* $\longrightarrow$
    if *next* $\neq$ *nullMId* $\longrightarrow$ *start_mission* . *next* $\longrightarrow$ *done_mission* . *next* $\longrightarrow X$
    $[]$ *next* $=$ *nullMId* $\longrightarrow$ **skip**
    fi
  Finish $\widehat{=}$ *end_sequencer_app* $\longrightarrow$ *end_mission_fw* $\longrightarrow$ *done_sequencer* $\longrightarrow$ **skip**
  $\bullet$ *Start* ; *Execute* ; *Finish*
end

FIGURE 9. Framework process for the mission sequencer.

next of type *MissionId* which contains identifiers for the missions of an application. Mission identifiers are introduced for all instances of classes of an application that extend the `Mission` infrastructure class. They allow us to uniquely identify objects of those classes. A special mission identifier *nullMId* is used to model the case when the method returns a Java `null` reference to signal that there are no more missions to execute.

Again, to illustrate the modelling approach, below we extract a parallel fragment that captures the behaviour that emerges from the composition of the mission sequencer framework and application processes.

$$\left(\begin{array}{l}\mu X \bullet \mathit{getNextMissionCall} \longrightarrow \\ \quad \mathit{getNextMissionRet}\,?\,\mathit{next} \longrightarrow A_{\mathit{cond}}(\mathit{next})\end{array}\right)$$
$$[\![\varnothing \mid \{\!|\, \mathit{getNextMission}[\mathit{Call}/\mathit{Ret}], \ldots \,|\!\} \mid \ldots]\!]$$
$$\left(\begin{array}{l}\ldots ; \\ \mathit{getNextMissionCall} \longrightarrow \mathbf{skip}; \\ \mathit{getNextMissionRet}\,!\,\mathit{ACCMId} \longrightarrow \mathbf{skip}\ ;\ \ldots\end{array}\right)$$

As before, the left action of the parallelism originates from the framework process, here the *Execute* action of *MissionSequencerFW* (whose conditional has been abbreviated by A_{cond}). The right action of the parallelism is provided by the mission sequencer application process *ACCMissionSequencerApp* as part of implementing the `getNextMission()` method. Notably, in this example, we have a synchronisation between an input prefix ($\mathit{getNextMissionRet}\,?\,\mathit{next} \longrightarrow \ldots$) and an output prefix ($\mathit{getNextMissionRet}\,!\,\mathit{ACCMId} \longrightarrow \ldots$). It results in a value being communicated, namely of a constant *ACCMId* of type *MissionId*. The value is locally bound by the input prefix to the variable *next* and hence can be accessed by the action A_{cond}.

In *Execute*, a conditional checks the value of *next*. If it is not equal to *nullMId*, synchronisations on *start_mission* . *next* and *done_mission* . *next* control the mission framework process *MissionFW* (defined below) that manages execution of the mission *next*, and then *Execute* recurses to handle the next mission. If *next* is equal to *nullMId*, *Execute* finishes. We note that semantically, there is no difference between a synchronisation $c\,.\,E \longrightarrow \mathbf{skip}$ and $c\,!\,E \longrightarrow \mathbf{skip}$ — they both result in outputting the value of E on channel c.

In the *Finish* action at the end, first a synchronisation on *end_sequencer_app* is used to terminate the mission sequencer application process. Next, a synchronisation on *end_mission_fw* terminates the mission framework process. Finally, synchronisation on *done_sequencer* acknowledges to the safelet process that the mission sequencer has finished.

4.3. Mission

The purpose of the mission framework process, *MissionFW*, is to record the mission's event handlers, execute the mission by synchronously starting these handlers, controlling their termination, and afterwards finishing the mission. Termination of a mission can be initiated by a handler at any point during the mission's execution phase, via a call to `requestTermination()`. The *MissionFW* framework process thus communicates with the mission sequencer process (*MissionSequencerFW*), the mission application processes, and the event handler processes.

Fig. 10 presents the definition of *MissionFW*. Its state has three components: the identifier *mission* of the mission being executed, if any, its finite set *handlers* of event handlers, and a flag *terminating* that records whether the current mission is in the process of termination. As previously mentioned, the handlers are identified by values of a type *HandlerId*. As with missions, we statically associate instances of classes that implement event handlers with unique identifiers.

The action *Init* defines that, initially, there is no mission executing, so that $\mathit{mission}'$ equals *nullMId*, and therefore, the set of handlers is empty. We also set *terminating* to *FALSE* reflecting that, to begin with, there is no termination request.

In the main action, we use again a modelling pattern where we have a sequence of actions that define the different phases of the entity life-cycle, as we already did in the safelet and mission sequencer models. In the case of the mission framework process, however, a recursion (operator $\mu X \bullet F(X)$, where occurrences of X in F are recursive invocations) perpetually calls this sequence of actions, because *MissionFW* controls

$$
\begin{array}{l}
\textbf{process}\ MissionFW \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ State == [mission : MissionId;\ handlers : \mathbb{F}\,(HandlerId);\ terminating : boolean] \\
\quad Init == [State' \mid mission' = nullMId \land handlers' = \varnothing \land terminating' = FALSE] \\
\quad Start \mathrel{\widehat=} Init\ ;\ start\_mission\,?\,m \longrightarrow mission := m \\
\quad AddHandler \mathrel{\widehat=} \textbf{val}\ handler : HandlerId \bullet handlers := handlers \cup \{handler\} \\
\quad Initialize \mathrel{\widehat=} initializeCall\,.\,mission \longrightarrow \\
\qquad \left(\mu X \bullet \begin{pmatrix} register\,?\,h \longrightarrow AddHandler(h)\ ;\ X \\ \Box \\ initializeRet\,.\,mission \longrightarrow \textbf{skip} \end{pmatrix}\right) \\
\quad StartHandlers \mathrel{\widehat=} |||\ h : handlers \bullet start\_handler\,.\,h \longrightarrow \textbf{skip} \\
\quad StopHandlers \mathrel{\widehat=} |||\ h : handlers \bullet stop\_handler\,.\,h \longrightarrow done\_handler\,.\,h \longrightarrow \textbf{skip} \\
\quad Execute \mathrel{\widehat=} StartHandlers\ ;\ activate\_handlers \longrightarrow \\
\qquad \begin{pmatrix} (stop\_handlers \longrightarrow StopHandlers\ ;\ done\_handlers \longrightarrow \textbf{skip}) \\ [\![\varnothing \mid \{\!|\, stop\_handlers, done\_handlers \,|\!\} \mid \{terminating\}]\!] \\ (Methods \triangle done\_handlers \longrightarrow \textbf{skip}) \end{pmatrix} \\
\quad Cleanup \mathrel{\widehat=} cleanupCall\,.\,mission \longrightarrow cleanupRet\,.\,mission \longrightarrow \textbf{skip} \\
\quad Finish \mathrel{\widehat=} end\_mission\_app\,.\,mission \longrightarrow done\_mission\,.\,mission \longrightarrow \textbf{skip} \\
\quad requestTerminationMeth \mathrel{\widehat=} \\
\qquad \begin{pmatrix} requestTerminationCall \longrightarrow \\ \textbf{if}\ terminating = FALSE \longrightarrow (terminating := TRUE\ ;\ stop\_handlers \longrightarrow \textbf{skip}) \\ [\!]\ terminating = TRUE \longrightarrow \textbf{skip} \\ \textbf{fi}\,; \\ requestTerminationRet \longrightarrow \textbf{skip} \end{pmatrix} \\
\quad terminationPendingMeth \mathrel{\widehat=} \\
\qquad terminationPendingCall \longrightarrow terminationPendingRet\,!\,terminating \longrightarrow \textbf{skip} \\
\quad Methods \mathrel{\widehat=} \mu X \bullet (requestTerminationMeth \ \Box\ terminationPendingMeth)\ ;\ X \\
\quad \bullet\ (\mu X \bullet Start\ ;\ Initialize\ ;\ Execute\ ;\ Cleanup\ ;\ Finish\ ;\ X) \triangle end\_mission\_fw \longrightarrow \textbf{skip} \\
\textbf{end}
\end{array}
$$

FIGURE 10. Mission framework process.

the execution of all missions in the program, and so repetitively offers its service. Termination of the mission cycle is forced by the mission sequencer process using the channel *end_mission_fw*.

The *Start* action initialises the state and waits for the mission sequencer to start a mission. Since the mission framework process can handle any mission, *Start* uses the channel *start_mission* to take a mission identifier m as an input, and records it in the state component *mission*. *Finish* uses that mission identifier to terminate the application process for the mission with a synchronisation on *end_mission_app . mission*, and to signal to the mission sequencer that the mission has finished with *done_mission . mission*. The channel parametrisations are necessary for the mission framework processes to establish a communication with particular missions: each mission application process synchronises only on the events for that mission.

The *Initialize* action models the initialisation phase, which is initiated by the framework by calling the `initialize()` method. It is specified using a recursion that continually accepts requests from the mission application process, through the channel *register*, to add a handler h to the mission; this is achieved by the parametrised action *AddHandler*. A synchronisation on *register* corresponds in the program to a call to the `register()` method of a handler class. The application process may besides use the event *initializeRet . mission* to terminate *Initialize* at any time. An external choice (operator $A_1 \ \Box\ A_2$) ensures that both communications are available, allowing the mission application process to exercise the choice.

The *Execute* action captures the main behaviour in executing a mission. It is a parallelism between an action that defines the control of handlers and an action that models infrastructure methods (*Methods* action) that are offered during execution. The parallel actions synchronise on the channels *stop_handlers* and *done_handlers*. These channels model the signal for stopping handler execution and the acknowledgement

$$
\begin{array}{l}
\textbf{process}\ \mathit{EventHandlerFW} \mathrel{\widehat=} h : \mathit{HandlerId} \bullet \textbf{begin} \\
\quad \textbf{state}\ \mathit{State} \mathrel{\widehat=} [\mathit{active} : \mathit{boolean}] \\
\quad \mathit{Init} \mathrel{\widehat=} [\mathit{State}' \mid \mathit{active}' = \mathit{FALSE}] \\
\quad \mathit{StartHandler} \mathrel{\widehat=} \mathit{start\_handler} \,.\, h \longrightarrow \mathit{active} := \mathit{TRUE} \\
\quad \mathit{ActivateHandler} \mathrel{\widehat=} \mathit{activate\_handlers} \longrightarrow \textbf{skip} \\
\quad \mathit{Start} \mathrel{\widehat=} (\mathit{StartHandler} \,;\ \mathit{ActivateHandler}) \,\Box\, \mathit{ActivateHandler} \\
\quad \mathit{DispatchHandler} \mathrel{\widehat=} \\
\qquad \mathit{enter\_dispatch} \,.\, h \longrightarrow \mathit{stop\_handler} \,.\, h \longrightarrow \\
\qquad \mathit{leave\_dispatch} \,.\, h \longrightarrow \mathit{done\_handler} \,.\, h \longrightarrow \textbf{skip} \\
\quad \mathit{Execute} \mathrel{\widehat=} \\
\qquad \textbf{if}\ \mathit{active} = \mathit{TRUE} \longrightarrow \mathit{DispatchHandler} \\
\qquad [\!]\ \mathit{active} = \mathit{FALSE} \longrightarrow \textbf{skip} \\
\qquad \textbf{fi} \\
\quad \bullet\ (\mu X \bullet \mathit{Init} \,;\ \mathit{Start} \,;\ \mathit{Execute} \,;\ X) \,\triangle\, \mathit{end\_mission\_fw} \longrightarrow \textbf{skip} \\
\textbf{end}
\end{array}
$$

FIGURE 11. Framework process for event handlers.

of handler termination. The name set of the left parallel action is empty, since it does not modify the state, whereas the right parallel action may set the *terminating* flag. The mission identifier and its handlers cannot be changed at this stage of mission execution.

As part of the control behaviour, first of all, all handlers are started with a call to the action *StartHandlers*, which uses synchronisations $\mathit{start\_handler} \,.\, h$ to start, all handlers h recorded in the state in interleaving. Each process corresponding to a handler h synchronises with the mission process on *start_handler*. The interleaving terminates after all synchronisations are performed.

The handlers do not immediately become active after they are started, that is, they do not respond to external events immediately. This is because they have to start execution synchronously, and the action *Start* uses a channel *activate_handlers* to ensure this. All handler processes synchronise on it, but only those that previously synchronised on *start_handler* proceed to execute their active behaviours. In this way, handlers can be registered asynchronously, but start execution (enter their dispatch loops) synchronously.

Termination of the handlers is initiated by the *requestTerminationMeth* action, with a synchronisation on *stop_handlers*. This action is one of the choices offered by the *Methods* action and corresponds to the `requestTermination()` method. Handler application processes call this method by synchronising on $\mathit{requestTermination}[\mathit{Call}/\mathit{Ret}]$. This causes the left parallel action in *Execute* to call the action *StopHandlers* after synchronising on *stop_handlers*. In case of multiple calls to `requestTermination()`, the *stop_handlers* event is only raised once. For each handler h of the mission, *StopHandlers* uses $\mathit{stop\_handler} \,.\, h$ to stop the handler, and then waits for the event $\mathit{done\_handler} \,.\, h$ acknowledging termination of the handler. Finally, when all handler processes of the mission have acknowledged termination, the interleaving in *StopHandlers* terminates, and the *done_handlers* event is raised. This last event forces the *Methods* action in the parallelism to be interrupted due to the interrupt action $\triangle\, \mathit{done\_handlers} \longrightarrow \textbf{skip}$, so that *Execute* altogether terminates, and the mission can proceed to the cleanup phase.

We note that the use of $\mathit{terminating} = \mathit{TRUE}$ and $\mathit{terminating} = \mathit{FALSE}$ instead of just *terminating* and $\neg\ \mathit{terminating}$ in the guards of the conditional is due to the fact that Z distinguishes between predicates and values. As a consequence, boolean values have to be introduced by virtue of a free type definition, such as $\mathit{boolean} ::= \mathit{TRUE} \mid \mathit{FALSE}$ in our model.

The framework process also supports method calls to `terminationPending()` during the execution phase. This method can be used by a handler to check if we are currently in the process of terminating a mission. It simply returns the value of *terminating*.

Finally, the *Cleanup* action calls the action of the mission application process corresponding to its `cleanup()` method. *Finish* is then invoked to report to the mission sequencer framework process that the current mission has terminated; this is via a synchronisation on $\mathit{done\_mission} \,.\, \mathit{mission}$.

The interrupt in the main action enables the mission sequencer to terminate the mission framework process via a synchronisation on *end_mission_fw*, namely when it is no longer needed. This happens during termination of the mission sequencer and, subsequently, the safelet.

4.4. Event Handlers

The framework process *EventHandlerFW* for an event handler is presented in Fig. 11. This process is the same

$$
\begin{array}{l}
\textbf{process}\ Framework \mathrel{\widehat=} handlers : \mathbb{F}\,(HandlerId) \bullet \\
\quad \Big(SafeletFW \\
\qquad [\![\{\!|\, start\_sequencer, done\_sequencer \,|\!\}]\!] \\
\quad \Big(\Big(MissionSequencerFW \\
\qquad [\![\{\!|\, start\_mission, done\_mission, end\_mission\_fw \,|\!\}]\!] \\
\quad \Big(\Big(MissionFW \\
\qquad [\![\{\!|\, start\_handler, stop\_handler, activate\_handlers, end\_mission\_fw \,|\!\}]\!] \Big) \\
\quad HandlersFW\,(handlers) \Big) \Big) \Big) \\
\qquad \setminus \{\!|\, start\_sequencer, done\_sequencer, start\_mission, done\_mission, \\
\qquad\quad start\_handler, stop\_handler, activate\_handlers \,|\!\}
\end{array}
$$

FIGURE 12. Framework process.

for periodic and aperiodic handlers. *EventHandlerFW* is parametrised by an identifier that must be provided when the framework process is instantiated for a particular handler. For the engine handler, for example, we use $EventHandlerFW\,(EngineHId)$.

The state component *active* of *EventHandlerFW* records whether the handler is active in the current mission or not. The main action defines an iterative behaviour that is interrupted and terminated by the event *end_mission_fw*, which, as mentioned before, indicates the end of mission execution.

Each iteration defines the behaviour of the handler during one mission. First, the state is initialised using *Init*, so that the handler is marked as inactive by default. Afterwards, the handler waits to be started, which is captured by the action *Start*. This action calls *StartHandler* in external choice with a synchronisation on *activate_handlers*, offered by *ActivateHandlers*. *StartHandler* synchronises on a particular *start_handler* event that is determined by the current handler identifier. Next, *StartHandler* also offers a synchronisation on *activate_handlers* (calling *ActivateHandlers*), which always occurs prior to entering the execution phase.

If the *start_handler* event of the handler occurs *before* *activate_handlers*, the value of *active* is *TRUE*. In this case, the behaviour of the handler, as defined by *Execute*, is to call *DispatchHandler*, which raises the $enter\_dispatch\,.\,h$ event of the particular handler h to notify the respective application process that it has to enter the dispatch loop. The dispatch loop can be interrupted after the $stop\_handler\,.\,h$ event, by synchronising on $leave\_dispatch\,.\,h$. The subsequent $done\_handler\,.\,h$ event notifies the mission framework process that the handler has terminated.

If *active* is *FALSE*, *Execute* terminates, as in that case the handler is not part of the current mission and remains idle during its execution. The subsequent recursive call ensures that the same handler can be run by multiple missions. Just like the mission process, a handler process perpetually offers its service until there are no more missions to be executed, and the mission framework terminates it by synchronisation on *end_mission_fw*, typically during the course of terminating the framework and application processes.

We observe that, unlike in the case of the safelet, the mission sequencer, and the mission framework processes, there is one instantiation of a handler framework process for each individual event handler of an application. The handler framework model is accordingly the parallel composition of all those process instances. For readability, we define a parametrised process *HandlersFW* that applies to a set of handler identifiers and yields the aforementioned composition.

$$
\begin{array}{l}
\textbf{process}\ HandlersFW \mathrel{\widehat=} handlers : \mathbb{F}\,(HandlerId) \bullet \\
\quad \|\ h : handlers \\
\qquad \{\!|\, activate\_handlers, end\_mission\_fw \,|\!\} \bullet \\
\qquad\quad EventHandlerFW\,(h)
\end{array}
$$

Here, we have an iterated parallelism over the handler identifiers h. We recapture that this is an iteration in the mathematical, and not the Java sense. It composes in parallel the processes $EventHandlerFW\,(h)$, where h ranges over the set *handlers* provided by the process parameter. All handlers synchronise on the channels *activate_handlers* and *end_mission_fw*. The handler framework processes essentially evolve independently, jointly synchronising only on the these two channels.

4.5. Overall SCJ Framework

To conclude, we define a process *Framework* that specifies the entire framework model. It is obtained by parallel composition of all framework processes. Like the *HandlersFW* process, the *Framework* process is parametrised by the handler identifiers for a particular application. For instance, in the ACC model they are *WheelShaftHId*, *EngineHId*, *BrakeHId*, *GearHId*, *LeverHId*, *SpeedoHId* and *ThrottleHId*.

The composite framework process is presented in Fig. 12. The control events that start or stop one of the SCJ components are hidden. Other framework events, like *end_mission_fw* are, however, exposed as we require application-level processes to synchronise on

Category	Type	Classes belonging to Category	Model
SMMC	active classes	Safelet, mission sequencer, and mission classes.	*Circus* process + *OhCircus* class
HC	active classes	Periodic and aperiodic event handler classes.	*Circus* process + *OhCircus* class
DC	data objects	Classes that model application data.	*OhCircus* class
IC	interaction classes	Classes for interaction with I/O devices.	—

TABLE 4. Encoding of different types of Java classes in the application model.

them. Table A.2 in Appendix A includes a summary of the channels that are associated with methods, including their parametrisation.

To illustrate the instantiation of *Framework* for a particular application, we define the corresponding framework process *ACCFW* for the ACC model.

$$\textbf{process}\ ACCFW \mathrel{\widehat=} Framework(\{WheelShaftHId, EngineHId, BrakeHId, GearHId, LeverHId, SpeedoHId, ThrottleHId\})$$

Here, we assume that *WheelShaftHId*, *EngineHId*, *BrakeHId*, and so on, have been introduced as unique identifiers of type *HandlerId*. The name of each constant is derived from the handler's class name. The constants themselves correspond to instances of a handler in a mission. For example, if we had two sensors and thus two instances of the `WheelShaft` class (counting the rotations of the wheel shaft), we would require two constants, $WheelShaftHId_1$ and $WheelShaftHId_2$, chiefly because in this case we require two separate handler framework processes.

So far, we have presented a formal model for the general SCJ programming paradigm in executing missions as realised by a compliant virtual machine. This model already provides useful insight into the exact mechanisms that underly the execution of Level 1 safelets, for instance, with regards to subtle details of the termination mechanism. Importantly, our model captures those mechanisms abstractly and independently of Java. In the next section, we discuss in detail how models for particular SCJ applications are constructed and expressed by a collection of *Circus* processes and *OhCircus* classes.

5. APPLICATION MODEL

Our presentation of the application model of SCJ is yet informal in this section and illustrated using the cruise controller. As before, we individually discuss the application processes for the safelet, mission sequencer, missions, and event handlers. We also discuss the model for data objects. The purpose of this section is to highlight the main principles and ideas of the modelling approach. In Section 6 we take these principles further by formalising the construction of models for arbitrary programs by virtue of a translation strategy, captured by a set of compositional translation rules.

5.1. Categories of Classes

For our modelling technique, we distinguish between four kinds of classes in an SCJ program (see Table 4). We consider

1. classes corresponding to the application's safelet, mission sequencer or missions;
2. classes corresponding to event handlers;
3. classes corresponding to data objects; and
4. classes for device interaction and I/O.

For readability, we refer to these categories by the names **SMMC**, **HC**, **DC**, and **IC**, respectively. The first and second categories cater for classes that either implement or extend an SCJ abstract class or interface for a Level 1 entity. For example, in the ACC we have `ACCSafelet`, `ACCMissionSequencer`, and `ACCMission` belonging to **SMMC**, and `Engine`, `Brake`, `ThrottleController`, and so on, belonging to **HC**. The third category comprises all other classes that do not belong to **IC**. Lastly, classes in **IC** implement interactions with external devices. They are not directly modelled and should, together with extra knowledge about the environment, justify assumptions regarding those interactions such as synchronicity, atomicity and instantaneity. These assumptions are explained in more detail in the next section. In the ACC, we have only the `Controller` class in **DC**. Classes in **IC** include interrupt service routines for hardware interrupts raised by the sensors.

We label the classes in **SMMC** and **HC** as active classes, since their models have to interact directly with framework processes by way of channel communications. Such active behaviour is, for instance, responding to calls to SCJ infrastructure methods that the application classes override. Therefore, **SMMC** and **HC** classes require process models. The classes in **DC** have *OhCircus* class models, thus they do not interact through communications with an active class. Generally, modelling SCJ classes as *OhCircus* classes has the advantage that we can treat their instances as values. On the other hand, in cases where data objects do interact with devices or the external hardware, we propose to refactor the code in a such a way that all interactions are moved into the handler classes; this seems to be usually possible in our experience so far.

$$
\begin{array}{l}
\textbf{process}\ \texttt{SCJClass}App\ \widehat{=}\ \textbf{begin} \\
\quad \textbf{state}\ State\ ==\ [this : \texttt{SCJClass}Class] \\
\quad Init\ \widehat{=}\ [State' \mid this' = \textbf{new}\ \texttt{SCJClass}Class] \\
\quad Meth_1\ \widehat{=}\ Meth_1 Call \longrightarrow (\ldots ;\ Meth_1 Ret \longrightarrow \textbf{skip}) \\
\quad Meth_2\ \widehat{=}\ Meth_2 Call \longrightarrow (\ldots ;\ Meth_2 Ret \longrightarrow \textbf{skip}) \\
\quad Meth_3\ \widehat{=}\ \ldots \\
\quad Methods\ \widehat{=}\ \mu X \bullet (Meth_1 \Box Meth_2 \Box \ldots) ;\ X \\
\quad \bullet\ Init ;\ (Methods \triangle end\_entity\_app \longrightarrow \textbf{skip}) \\
\textbf{end}
\end{array}
$$

FIGURE 13. Basic process model for the safelet, mission sequencer and mission application classes.

A general basic pattern for the process model of the classes in **SMMC** is presented in Fig. 13. The name of the process is derived from the SCJ class by appending the class name with *App*. The state of the process contains a single component *this* whose type is that of an *OhCircus* class. This class is in direct correspondence with the respective SCJ class. Whereas the process `SCJClass`*App* models calls to infrastructure methods, the class `SCJClass`*Class* models the actual data object and non-infrastructure methods that do not require interaction with the framework or external devices. We can hence think of the process as 'wrapping' the class object. Fig. 13 describes the general wrapping pattern.

The behaviour of an application process is first to initialise the state component *this* with a new *OhCircus* data object (using the action *Init*) and then to offer calls to all infrastructure methods (using the action *Methods*), until a communication *end_entity_app* occurs that terminates the process. The actual name of this channel depends on the particular SCJ component. For instance, for the safelet it is *end_safelet_app*, and for the mission sequencer, *end_sequencer_app*.

If the SCJ class does not have any instance variables, the corresponding process can be simplified by removing the state paragraph and the *Init* action. Constructors of the SCJ class are modelled in the *OhCircus* class, by *OhCircus* constructors. For now, we assume the presence of only parameterless constructors in **SMMC** classes (safelet, mission sequencer, and mission classes). Handler classes (in **HC**), as discussed later on, have a different model and can make unconstrained use of parametrised constructors.

Specific method actions $Meth_1$, $Meth_2$, and so on, must be present depending on the SCJ component. Table A.2 in Appendix A indicates the methods that are required in each class. The choice offered by *Methods* is exercised by the associated framework process or other application processes that call those methods. We note that method calls in our model have to respect the behavioural restrictions of SCJ [2]. That is, certain methods should only be called during the initialisation or finalisation phase of the safelet or of a mission.

A method action $Meth_i$ in Fig. 13 synchronises on the channel that represents a call to the method and then executes actions that correspond to the method implementation, indicated by the dots. If the method is a pure data operation, we invoke the *OhCircus* model of the method using a call $this.Meth_i(args)$. If the method interacts through communications, we embed a model of the method as an action directly into the process. Before termination, the method action synchronises on the $Meth_i Ret$ channel to signal the return of the call. As previously noted, for methods that have parameters or return a value, the call and return channels are used to communicate these values. In that case, the shape of the corresponding $Meth_i$ action is more elaborate, following the pattern illustrated below.

$$
Meth_i\ \widehat{=}\ \left(\begin{array}{l} Meth_i Call ?\ args \longrightarrow \\ \left(\begin{array}{l} \textbf{var}\ ret : T \bullet MBody_i(args, ret); \\ Meth_i Ret\,!\ ret \longrightarrow \textbf{skip} \end{array} \right) \end{array} \right)
$$

Instead of a simple synchronisation, the call is modelled by an input prefix that receives the arguments *args* of the call. The local variable *ret* holds the return value and is initialised by the action $MBody_i$. The return value is communicated as an output through the return channel $Meth_i Ret$, which is also parametrised.

The basic model in Fig. 13 assumes that there is only a single instance created for each class. This is the case for the safelet and mission sequencer, however the same mission class may potentially be instantiated more than once by the mission sequencer. To support multiple instances, we modify the main action of the basic application process as shown below.

$$
\begin{array}{l}
(\mu X \bullet Init ;\ Methods \triangle end\_entity\_app \longrightarrow X) \\
\qquad \triangle\ end\_mission\_fw \longrightarrow \textbf{skip}
\end{array}
$$

Here, raising the *end_entity_app* event causes the application process to be restarted rather than its termination because of the recursive call to X. The framework event *end_mission_fw* is used instead to terminate the process during safelet shutdown.

We next discuss the application model for the various infrastructure classes of the ACC in detail.

$$
\begin{array}{l}
\textbf{process}\ ACCSafeletApp \mathrel{\widehat=} \textbf{begin} \\
\quad setUpMeth \mathrel{\widehat=} \\
\qquad \left(\begin{array}{l} setUpCall \longrightarrow \textbf{skip}; \\ setUpRet \longrightarrow \textbf{skip} \end{array}\right) \\
\quad tearDownMeth \mathrel{\widehat=} \\
\qquad \left(\begin{array}{l} tearDownCall \longrightarrow \textbf{skip}; \\ tearDownRet \longrightarrow \textbf{skip} \end{array}\right) \\
\quad Methods \mathrel{\widehat=} \mu X \bullet (setUpMeth \,\Box\, tearDownMeth)\ ;\ X \\
\quad \bullet\ Methods \ \triangle\ end\_safelet\_app \longrightarrow \textbf{skip} \\
\textbf{end}
\end{array}
$$

FIGURE 14. Application process for the safelet of the cruise controller.

$$
\begin{array}{l}
\textbf{process}\ ACCMissionSequencerApp \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ State == [this : ACCMissionSequencerClass] \\
\quad Init == [State' \mid this' = \textbf{new}\ ACCMissionSequencerClass] \\
\quad getNextMissionMeth \mathrel{\widehat=} getNextMissionCall \longrightarrow \\
\qquad \left(\begin{array}{l} \textbf{var}\ ret : MissionId \bullet this.getNextMission(ret); \\ getNextMissionRet\,!\,ret \longrightarrow \textbf{skip} \end{array}\right) \\
\quad Methods = \mu X \bullet getNextMissionMeth\ ;\ X \\
\quad \bullet\ Init\ ;\ (Methods \ \triangle\ end\_sequencer\_app \longrightarrow \textbf{skip}) \\
\textbf{end}
\end{array}
$$

FIGURE 15. Application process for the mission sequencer of the cruise controller.

5.2. Safelet

We present the application process for the ACC safelet in Fig. 14. It is in direct correspondence with the `ACCSafelet` class. The specification is trivial here since `setUp()` and `tearDown()` in `ACCSafelet` do not contain any code (the actions are just **skip**). The process nevertheless illustrates the modelling approach for SCJ classes belonging to the category **SMMC**. The actions offered by *Methods* are *setUpMeth* and *tearDownMeth*. Termination occurs when the safelet framework process raises the *end_safelet_app* event. The process definition follows the general pattern in Fig. 13, however, simplifications have been possible as the resulting process lacks a state and an initialisation.

5.3. Mission Sequencer

The mission sequencer application process is given in Fig. 15. It more completely illustrates our approach to modelling SCJ classes as *OhCircus* classes and *Circus* processes. *ACCMissionSequencerApp* utilises the definition of the *OhCircus* class in Fig. 16. The *OhCircus* class is in direct correspondence with the `ACCMissionSequencer` SCJ class, shown in Fig. 17. The instance variables of the SCJ class become state components of the *OhCircus* class. Here, we have only one state component *mission_done*, corresponding to a variable of the same name in the SCJ class. The action *Init* that follows specifies the constructor behaviour.

Methods that are called by the SCJ infrastructure are defined in both, the *Circus* process and *OhCircus* class, whereas methods that are not called by the infrastructure are defined in the *OhCircus* class only if they are data operations. A special case are methods that interact with external devices. Those methods have mere action models, defined in the process.

The method `getNextMission()`, for instance, is called by the infrastructure, so it has to be specified in the application process too (Fig. 15). In the class definition of the method, we have a conditional that, depending on the value of *mission_done* returns the next mission whose identifier is either *ACCMId* or *nullMId*. The result parameter *ret* of type *MissionId* is introduced to hold the return value. The action model of the method synchronises on the respective *Call* channel to wait for a call from the framework, then invokes the method on the aggregated *this* class object, and finally synchronises on the *Ret* channel to finalise the method call and communicate its result.

In general, we model methods of SCJ classes by *OhCircus* methods where this is possible: methods that perform data operations become *OhCircus* methods, whereas methods that are called by the framework or interact with external devices become actions.

$$
\begin{array}{l}
\textbf{class}\ ACCMissionSequencerClass \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ State == [\textbf{private}\ mission\_done : boolean] \\
\quad \textbf{initial} \mathrel{\widehat=} mission\_done := false \\
\quad \textbf{public}\ getNextMission \mathrel{\widehat=} \textbf{res}\ ret : MissionId \bullet \\
\qquad \textbf{if}\ mission\_done = FALSE \longrightarrow \\
\qquad\quad (mission\_done := TRUE\ ;\ ret := ACCMId) \\
\qquad [\!]\ mission\_done = TRUE \longrightarrow ret := nullMId \\
\qquad \textbf{fi} \\
\textbf{end}
\end{array}
$$

FIGURE 16. *OhCircus* class for the mission sequencer of the cruise controller.

```
public class ACCMissionSequencer extends MissionSequencer {
  /* Records if the mission has already been executed. */
  private boolean mission_done;

  public ACCMissionSequencer() {
    super(...);
    mission_done = false;
  }

  public Mission getNextMission() {
    if (!mission_done) {
      mission_done = true;
      return new ACCMission();
    }
    else {
      return null;
    }
  }
}
```

FIGURE 17. Java class of the mission sequencer of the cruise controller.

5.4. Mission

For mission application processes, we use the modified form of the basic process model in Fig. 13 as discussed in Section 5.1. The main action of the *ACCMission* process accordingly has the following shape.

$$
\begin{array}{l}
\Big(\mu X \bullet Init; \\
\quad \big(Methods \triangle end\_mission\_app\,.\,ACCMId \longrightarrow X\big)\Big) \\
\qquad \triangle\ end\_mission\_fw \longrightarrow \textbf{skip}
\end{array}
$$

The *end_mission_app* channel is parametrised to enable the termination of particular mission application processes. Here, there is only one mission process which synchronises on *end_mission_app* . *ACCMId*.

Java methods are encoded as before by a combination of *OhCircus* class methods and local actions. Notably, the encoding of `initialize()` is only possible as an action due to its communication with the framework when registering handlers. Thus, there is no model of this method in the underlying *OhCircus* class. Generally, `initialize()` creates objects for shared data and handlers, and registers those handlers with the framework, which subsequently executes them as part of the current mission. Below, we include an extract of its action model for the ACC implementation.

$$
\begin{array}{l}
initializeMeth \mathrel{\widehat=} initializeCall\,.\,ACCMId \longrightarrow \\
\Bigg(\begin{array}{l}
\textbf{var}\ \ldots;\ speedo : SpeedMonitorClass; \\
\qquad throttle : ThrottleControllerClass; \\
\qquad cruise : ControllerClass; \\
\qquad engine : EngineClass;\ \ldots \bullet \\
\ldots; \\
throttle := \textbf{new}\ ThrottleControllerClass(speedo); \\
ThrottleControllerInit\,!\,throttle \longrightarrow \textbf{skip}; \\
register\,.\,ThrottleHId \longrightarrow \textbf{skip} \\
cruise := \textbf{new}\ ControllerClass(throttle, speedo); \\
engine := \textbf{new}\ EngineClass(cruise); \\
EngineInit\,!\,engine \longrightarrow \textbf{skip}; \\
register\,.\,EngineHId \longrightarrow \textbf{skip}; \\
\ldots; \\
initializeRet\,.\,ACCMId \longrightarrow \textbf{skip}
\end{array}\Bigg)
\end{array}
$$

The SCJ code of this method can be found in Fig. 4. Like in the program, local variables are introduced for handler objects, such as *speedo*, *throttle*, and *engine*. These variables have a class type, and are initialised with newly created class objects using a

```
class EngineClass ≙ begin
  state EngineState == [private cruise : ControllerClass]
  initial EngineInit ≙ val c : ControllerClass • cruise := c
  public handleAsyncLongEvent ≙ val evt : long •
    if evt = EngineOn ⟶ cruise.engineOn()
    [] evt = EngineOff ⟶ cruise.engineOff()
    fi
end
```

FIGURE 18. *OhCircus* class for the `Engine` handler.

suitable constructor. Our modelling approach requires that handler objects are only created and registered inside this method. This is enforced by the behavioural restrictions of SCJ [2]. The two synchronisations on the channels *ThrottleControllerInit* and *EngineInit* link the handler processes to their underlying data objects. Specifically, the process *ThrottleControllerApp* synchronises on *ThrottleControllerInit* and the process *EngineApp* on *EngineInit*. Hence the instantiation of a handler in the program such as, for instance,

```
engine = new Engine(cruise);
```

is modelled by a sequence of two statements

$$engine := \mathbf{new}\ EngineClass(cruise);$$
$$EngineInit\,!\,engine \longrightarrow \mathbf{skip}$$

where the assignment is implicitly a reference assignment to an object of (class) type *EngineClass*. The *cruise* object is an object of the class *Controller* that is shared between the handlers, hence it does not have a process model, nor do we require a synchronisation on a channel *Init* to model its creation.

The *Init* channels are introduced for all handler classes, but they are not needed for the safelet, mission sequencer and mission classes. This is due to the fact that for the latter classes, the link between a process and the underlying data object is static with respect to the model, whereas in case of handlers it is established dynamically during initialisation of the mission.

To record a periodic or aperiodic handler as part of the current mission, we have a communication such as $register\,.\,EngineHId \longrightarrow \mathbf{skip}$. In the program, this corresponds to a call to the `register()` method of the handler classes of the SCJ infrastructure.

The inherited methods of `ACCMission`, namely `requestTermination()` and `terminationPending()`, are provided by the *MissionFW* framework process since they cannot be overridden. The `cleanup()` method of `ACCMission` just unregisters ISRs, and hence is not discussed here.

Our modelling approach requires that there is a one-to-one relationship between instances of SCJ classes and their processes at any given time. For handlers, this assumption is justified if we consider a particular handler only to be instantiated once per mission. To cater for multiple instances of the same handler, multiple instances of both the underlying handler framework and application processes are necessary.

We recapture that not all classes of an SCJ program require process models. In particular, those belonging to **DC** have mere *OhCircus* class models, such as the `Controller` class. Conversely, SCJ components without any instance fields do not require a class model and thus have only a process model.

5.5. Event Handlers

Handler classes belong to the category **HC** and, as already noted, their application processes differ in terms of structure from those of the safelet, mission sequencer and missions (belonging to category **SMMC**). Most significantly, the application process for a handler may associate external events to it. In addition, it has a dispatch action *Dispatch* to release the handler either when one of these events occurs, or periodically in the case of periodic handlers. The general pattern, however, is still that described in Fig. 13.

As with classes belonging to the category **SMMC**, the application processes for handlers are factored into a data object modelled by an *OhCircus* class, and a process that aggregates the data object and releases the handler. Because we treat handlers as data objects, other data objects can hold a reference to them and directly call their methods or access and modify their fields. In the cruise controller, for instance, the `Controller` class holds a reference to the `SpeedMonitor` and `ThrottleController` handlers and thereby is able to acquire the vehicle's speed and set the throttle voltage. Since `Controller` is a data object (not wrapped by a process), these calls can be modelled by *OhCircus* method calls rather than synchronisations. This simplifies the overall application model.

Fig. 18 presents the *OhCircus* class for the `Engine` SCJ class, included in Appendix B.1. As before, we have a direct correspondence with instance variables defined as state components, and the constructor defined in the **initial** paragraph. The only difference

$$
\begin{array}{l}
\textbf{process}\ \mathit{EngineApp} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{EngineState} == [\mathit{this} : \mathit{EngineClass}] \\
\quad \mathit{Init} \mathrel{\widehat=} \mathit{EngineInit}\,?\,\mathit{obj} \longrightarrow \mathit{this} := \mathit{obj} \\
\quad \mathit{handleAsyncEventMeth} \mathrel{\widehat=} \textbf{val}\ \mathit{evt} : \mathit{EventId} \bullet \\
\qquad \textbf{wait}\ 0\,..\,\mathit{EngineBudget}\ ;\ \mathit{this}.\mathit{handleAsyncLongEvent}(\mathit{evt}) \\
\quad \mathit{Execute} \mathrel{\widehat=} \mathit{enter\_dispatch}\,.\,\mathit{EngineHId} \longrightarrow \mathit{Dispatch} \\
\quad \mathit{Dispatch} \mathrel{\widehat=} \\
\qquad \left(\mu X \bullet \left(\begin{array}{l} \mathit{engine\_on} \longrightarrow \mathit{handleAsyncEventMeth}(\mathit{EngineOnEvtId})\ ;\ X \\ \Box \\ \mathit{engine\_off} \longrightarrow \mathit{handleAsyncEventMeth}(\mathit{EngineOffEvtId})\ ;\ X \\ \Box \\ \mathit{leave\_dispatch}\,.\,\mathit{EngineHId} \longrightarrow \textbf{skip} \end{array} \right) \right) \\
\quad \bullet\ (\mu X \bullet \mathit{Init}\ ;\ \mathit{Execute}\ ;\ X) \triangle \mathit{end\_mission\_fw} \longrightarrow \textbf{skip} \\
\textbf{end}
\end{array}
$$

FIGURE 19. Application process for the `Engine` handler.

is that to model the static fields `Events.EngineOn` and `Events.EngineOff`, we use the constants *EngineOn* and *EngineOff*. The `Events` class provides unique `long` values for all external events of the cruise controller as a collection of public, static and final fields. Although we do not consider models of static fields and methods in general, a `static` and `final` field that is initialised upon declaration is modelled by introducing a global axiomatic constant of the same name. (Issues related to name clashes, if present, are dealt with by prefixing the name of the constant with the name of the class in which the field resides in the program.)

The handler method is parametrised, mirroring the parameter of `handleAsyncLongEvent(int)`. As explained in Section 2.2, the parameter is used to determine the external event that caused the release of the handler. Based on its value, the handler method decides by virtue of a `switch` statement which method to call on the controller object, and this is modelled by a conditional in Fig. 18. The classes used to implement the interrupt service routines that identify external events and release the corresponding handlers belong to the category **IC** and are only interesting to the model in as far as they guarantee that the event identifier passed is indeed the one of the last event raised. How this is achieved (perhaps in terms of low-level instructions) is not a concern here, but we discuss this issue further in Section 7.

The process for the *Engine* handler is presented in Fig. 19. As described in the wrapping pattern in Fig. 13, the object for the handler is recorded in a state component *this*. The *Init* action initialises this component when prompted by a communication on the designated channel *EngineInit* of type *EngineClass* that provides an object as input. The creation of the data object that is aggregated by the process takes place outside the process, namely in the *initializeMeth* action of the *ACCMission* process. Since a handler may be used by several missions, the application process repeatedly initialises (*Init*) and executes (*Execute*) the handler in a recursive action.

The *Execute* action waits for the *enter_dispatch* event of the handler to occur, and then calls the action *Dispatch*, which enters a dispatch loop that repeatedly waits for the occurrence of one of the external events associated with the handler. In our example, they are *engine_on* and *engine_off*. When such an event occurs, *Dispatch* calls the *handleAsyncEventMeth* action, passing an input value identifying the event. It also offers a synchronisation on *leave_dispatch*, which is used to abandon the dispatch loop when the handler is terminated by its associated framework process.

The *handleAsyncEventMeth* action simply executes the corresponding data operation, preceded by a nondeterministic wait that sets a time budget: the permissible amount of time the program may take to execute the operation. Abstract global constants are introduced for all data operations to refer to their worst-case execution time. Since *OhCircus* method calls are instantaneous, all timing behaviour is specified explicitly within actions using these constants.

When `handleAsync[Long]Event()` raises an output event, it has to be modelled in a different way. For example, the handler process for the throttle controller has to perform communications $\mathit{set\_voltage}\,!\,v$ that correspond to a device access that takes place inside the `writeVoltage()` method. Here, we cannot represent the method by a data operation as above, but have to encode `writeVoltage()` as an action as illustrated in Fig. 20. The *handleAsyncEventMeth* action of the application process reflects the Java code, but outputs a value where in the SCJ program we have specific instructions for writing device data.

A consequence of our modelling approach is that

$$
\begin{array}{l}
\textbf{process}\ \mathit{ThrottleControllerApp} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{ThrottleControllerState} == [\mathit{this} : \mathit{ThrottleControllerClass}] \\
\quad \mathit{Init} \mathrel{\widehat=} \mathit{ThrottleControllerInit}\,?\,\mathit{obj} \longrightarrow \mathit{this} := \mathit{obj} \\
\quad \mathit{writeVoltageMeth} \mathrel{\widehat=} \mathit{set\_voltage}\,!\,\mathit{this.voltage} \longrightarrow \textbf{skip} \\
\quad \mathit{handleAsyncEventMeth} \mathrel{\widehat=} \\
\qquad \textbf{if}\ \mathit{this.scheduleThrottle} = \mathit{TRUE} \longrightarrow \\
\qquad\quad \textbf{if}\ \mathit{this.accelerating} = \mathit{TRUE} \longrightarrow \\
\qquad\qquad (\textbf{wait}\ 0\,..\,\mathit{increaseVoltageBudget}\ ;\ \mathit{this.increaseVoltage}()\ ;\ \mathit{writeVoltageMeth}) \\
\qquad\quad [\!]\ \mathit{this.accelerating} = \mathit{FALSE} \longrightarrow \ldots \\
\qquad\quad \textbf{fi} \\
\qquad [\!]\ \mathit{this.scheduleThrottle} = \mathit{FALSE} \longrightarrow \textbf{skip} \\
\qquad \textbf{fi} \\
\quad \mathit{Execute} \mathrel{\widehat=} \mathit{enter\_dispatch}\,.\,\mathit{ThrottleHId} \longrightarrow \\
\qquad (\mathit{Dispatch} \llbracket \{\mathit{this}\} \mid \{\!|\ \mathit{release\_handler}\ |\!\} \mid \varnothing \rrbracket \mathit{Release}) \setminus \{\!|\ \mathit{release\_handler}\ |\!\} \\
\quad \mathit{Dispatch} \mathrel{\widehat=} \\
\qquad \left(\mu X \bullet \begin{pmatrix} \mathit{release\_handler}\,.\,\mathit{ThrottleHId} \longrightarrow \mathit{handleAsyncEventMeth}() \\ \Box \\ \mathit{leave\_dispatch}\,.\,\mathit{ThrottleHId} \longrightarrow \textbf{skip} \end{pmatrix} ;\ X\right) \\
\quad \mathit{Release} \mathrel{\widehat=} \\
\qquad \left(\mu X \bullet \begin{pmatrix} \mathit{release\_handler}\,.\,\mathit{ThrottleHId} \longrightarrow \textbf{skip}; \\ \textbf{wait}\ \mathit{this.period} \end{pmatrix} ;\ X\right) \\
\qquad\quad \triangle\ \mathit{leave\_dispatch}\,.\,\mathit{ThrottleHId} \longrightarrow \textbf{skip} \\
\quad \bullet\ (\mu X \bullet \mathit{Init}\ ;\ \mathit{Execute}\ ;\ X) \triangle \mathit{end\_mission\_fw} \longrightarrow \textbf{skip} \\
\textbf{end}
\end{array}
$$

FIGURE 20. Application process for the `ThrottleController` handler.

writeVoltageMeth in Fig. 20 cannot be invoked by an *OhCircus* method. In the cruise controller this is not a problem, since no other method except for `handleAsyncEvent()` calls it. This, however, hints a more general issue: methods that carry out some device access inherently require an action model, and so do all methods that directly or indirectly call them. Our technique hence imposes certain restrictions on the SCJ program designs that we can cater for, and Section 6 examines those restrictions in more detail.

Application processes for periodic handlers differ only in the definition of *Execute* and in the presence of an additional action *Release*. As illustrated in Fig. 20, the handler behaviour in this case is defined by a parallelism between two actions *Dispatch* and *Release*. Here, the handler does not wait for the occurrence of an external event, but instead invokes *handleAsyncEventMeth* when an internal handler-specific timer event is raised.

The parallel action *Release* generates these timer events. In its definition in Fig. 20, we have a prefix that raises the *release_handler . ThrottleHId* event, which is the timer event for the throttle handler, followed by a **wait** *this.period* action to wait for the duration of the period. The recursion in *Release* ensures that timer events are generated continually until an interrupt occurs, causing the handler to leave its dispatch loop. If *Dispatch* is still executing *handleAsyncEventMeth* when *Release* is ready again to synchronise on *release_handler . h*, the synchronisation is delayed until the handler method is completed. Since the timer event is concealed (hidden) in the system model, it takes place autonomously and in fact as soon as possible, that is, when both *Dispatch* and *Release* are ready to synchronise on it.

To increase modularity, we take advantage of class inheritance by introducing a class for periodic handlers that has a state component *period*. This class somewhat corresponds to the abstract class `PeriodicEventHandler` in SCJ. We thus require that concrete periodic handler classes extend this class also in the model. The period is set by either passing a value to the constructor of the base class, or explicitly by changing the value of *period* in the subclass constructor.

We observe that the handler application processes in our example lack a *Methods* action. This is because in the ACC, the handlers do not define methods called by the framework. Furthermore, the call to `handleAsync[Long]Event([int])` takes place within the process and is modelled by an action call. In general, however, we do not exclude the possibility of defining such methods, following the pattern in Fig. 13.

class *ControllerClass* $\widehat{=}$ **begin**

state *ControllerState*

$throttle : ThrottleControllerClass$
$speedo : SpeedMonitorClass$
$engineActive : boolean$
$braking : boolean$
$\ldots$

initial *ControllerInit*

$ControllerState'$
$throttle? : ThrottleControllerClass$
$speedo? : SpeedMonitorClass$

$throttle' = throttle? \land speedo' = speedo?$
$engineActive' = FALSE \land braking' = FALSE \land \ldots$

public sync *engineOn* $\widehat{=}$
$engineActive := TRUE \ ; \ braking := FALSE \ ; \ topGear := FALSE \ ; \ cruising := FALSE$

public sync *activate* $\widehat{=}$
if $engineActive = TRUE \land topGear = TRUE \land braking = FALSE \longrightarrow$
$\left(\begin{array}{l} cruising := TRUE; \\ \textbf{var } cruise\_speed : int \bullet \\ cruise\_speed := speedo.getCurrentSpeed(); \\ throttle.setCruiseSpeed(cruise\_speed); \\ throttle.schedule(); \\ \ldots \end{array}\right)$
$[] \neg (engineActive = TRUE \land topGear = TRUE \land braking = FALSE) \longrightarrow$ **skip**
fi

public sync *deactivate* $\widehat{=}$
if $engineActive = TRUE \land topGear = TRUE \land cruising = TRUE \land \ldots$

end

FIGURE 21. *OhCircus* class definition for the `Controller` SCJ class.

5.6. Data Objects

As already noted, data objects are modelled by an *OhCircus* class only. For illustration, Fig. 21 contains an extract from the *OhCircus* class definition for the `Controller` SCJ class. We observe that since this class holds a reference to two handler objects, we have state components of type *SpeedMonitorClass* and *ThrottleControllerClass*. All methods in this class are declared as synchronised using the *OhCircus* keyword **sync** explained in Section 2.3.2. Some state components and methods have been omitted or abbreviated for brevity.

The `Controller` class in essence implements a state machine. The methods are called by aperiodic handlers in response to external events such as the engine being switched on or the cruise mode being activated. For instance, the *engineOn*() method changes the values of some of the state components of *boolean* type. More interestingly, inside the *activate*() method we have calls to methods on the aggregated *throttle* and *speedo* class objects. These calls directly access and modify the data (state) of the class objects of the respective handlers.

To conclude, in the previous sections we have examined in detail the construction of a formal model for an SCJ program, elaborating and refining our account in [37]. In particular, we have unified the treatment of behavioural and data aspects by way of the general pattern in Fig. 13. In the next section, we formalise a translation strategy that constructs *Circus* models of SCJ programs using the approach described.

6. TRANSLATION STRATEGY

Given an SCJ program, the problem we address in this section is the automatic derivation of its application model. Not all SCJ programs are translatable by our approach, and, so far, we have been vague about the subset of SCJ that we handle, as well as the details of the model construction process. We now present a compositional translation strategy.

In Section 6.1, we first specify the admissible subset of SCJ programs. Section 6.2 then discusses the top-level translation process. The remaining Sections 6.3

to 6.5 examine in detail the translation of SCJ classes for the entities of the mission model and data objects.

6.1. Translatable Programs

The programs we accept foremost have to be compilable with a compliant SCJ class library. Since SCJ is still evolving, we consider the API published in the first official draft of the SCJ technology specification [2]. The generality of Java, however, enables us to write SCJ programs that do not reflect the clean architecture implied by the SCJ paradigm. To give an example, the same class can be used as the `Safelet` and `MissionSequencer` of an application. We disallow such designs and more generally require programs to adhere to design patterns that embody good programming practice. Finally, as already noted in the introduction, the design of SCJ [2] does not address constraints on statements that make certification (and formal analysis) feasible. We address this concern by taking restrictions of SPARK [32] as a guideline. We note that checking for compliance with the restrictions we introduce can be automated in a fairly straightforward manner.

In the sequel, we specify our restrictions in more detail. We divide them into three kinds: structural constraints, language constraints and feature constraints.

6.1.1. Structural Constraints

Our first structural constraint, as hinted above, is that the safelet, the mission sequencer, and each mission and event handler are modelled by a separate class. This ensures that there is a one-to-one correspondence between classes that represent components of the SCJ infrastructure and their respective process descriptions in the application model. Applications that initially do not satisfy this constraint can usually be refactored.

A second structural constraint is that we only consider one level of inheritance in the context of the SCJ classes. This means that the safelet, mission sequencer, mission and handler classes of an application have to be direct subclasses of the respective SCJ API class or interface. The rationale for this restriction is that, because these classes belong to either category **SMMC** or **HC**, they have a process model, and there is currently no notion of process inheritance in *OhCircus*. Other classes, namely those belonging to category **DC** can make unconstrained use of inheritance (class inheritance is well supported in *OhCircus*).

A third constraint is concerned with the places in the program where we allow interaction with the hardware and external devices. Confining external device interaction to the handler classes enforces good programming practice, resulting in cleaner application designs with more tractable formal models. We thereby obtain a clean separation of active classes that interact with the framework, having process models, and passive classes that record application data and carry out computational work. This may possibly be at the cost of reducing reusability, but, what is important, facilitates analysis and the refinement of models.

The fourth and last constraint is that we prohibit the use of inner and anonymous classes. This is not a serious limitation since such constructs are merely syntactic sugar and can always be eliminated through refactoring. Excluding them discharges the burden of having to define a semantics for them in *OhCircus* which, by default, does not cater for such classes.

Together the four constraints determine the structure of applications for which model generation succeeds.

6.1.2. Language Constraints

As already mentioned, the SCJ technology specification [2] does not prescribe what low-level language elements and constructions are permitted in SCJ programs. In a certification context, it is, however, not desirable to support the full generality of Java. Our language restrictions are similar in spirit to the ones defined by SPARK [32]. They do aid the clarity and analysability of SCJ programs, but ultimately account also for limitations implied by our translation strategy.

The following list summarises the elements of the Java language that we exclude.

1. Expressions with side effects such as `(x++)*(y--)`. This applies to any type of expression, namely boolean expressions in conditional statements and loops. Likewise, none of the arguments of a method call may have side effects either.
2. Labels as well as stand-alone `break` and `continue`.
3. Arbitrarily placed `returns`: a `return` statement must be the last statement in a method, if present.
4. Fall-through behaviour in `switch` statements. This means that every `case` statement must be properly terminated with a `break` statement.
5. Program exceptions and hence the use of `try`, `throw` and `catch` and, in general, also `finally`.
6. Blocks that are marked as `synchronized` as well as the `wait()` and `notify()` methods.
7. `static` methods; `static` fields are only supported if they are `final` and initialised upon declaration.

From (1), it implicitly follows that we do not support assignments in expressions. We observe though that expressions with side effects can always be rewritten into statements to fulfil this constraint: this is by way of introducing local variables that hold intermediate results of a calculation. We also point out that (1) does not exclude assignment statements as stand-alone commands. For instance, `x = x + y;` and `x++;` are allowed as they do not occur in an expression.

The restrictions (2) and (3) also exist in safe subsets of Ada and C [32]. The precise reasons for them in terms of our modelling approach are discussed later on (page 28). The absence of exceptions (5) is related to issues of memory utilisation and predictability in SCJ. Although exceptions can be useful in designing error catching mechanisms, they may be less essential

where formal verification techniques are consequently applied. A limited facility to model exceptions that raise program errors is provided by the divergent action **abort**. We observe that the blocking constructs in (6) are already excluded by SCJ Level 1. Support for static methods (7) is work in progress; `final` and `static` fields, on the other hand, can effectively be treated as constants and are thus already incorporated.

6.1.3. Feature Constraints

Some SCJ Level 1 classes are not covered by our translation strategy. These classes are typically either an artifact of the implementation of SCJ or of program interaction with devices and the external world. Excluded classes are in particular:

1. Classes such as `ManagedEventHandler`, which are an artifact of SCJ's design on top of RTSJ.
2. The `Clock` class and all timing-related classes except for `RelativeTime` and `AbsoluteTime`.
3. Classes of the input and output model, like, for example, `ConsoleConnection`.
4. The Java Native Interface (JNI).

To explain (1), we note that the original aim of SCJ was to design it as a restriction of the RTSJ class library. The RTSJ classes are, however, not directly relevant to SCJ programs. They merely facilitate the development of a reference implementation of the SCJ technology on top of RTSJ. (There are also inclinations in the SCJ community to depart from this design in future versions of SCJ, hence it is not desirable to model them here.)

The features (2) and (3) are in principle desirable to support as future work, but with regards to our current contribution out of scope. The concession we make for not supporting (2) is that we exclude applications that make more sophisticated use of time control. The classes `AbsoluteTime` and `RelativeTime` are nevertheless supported by way of data objects. They are used to set the period of periodic handlers.

Support for (3) seems not difficult to achieve, but is not essential for our case studies. In our experience so far, there also seems to be no need for (4). This is due to the abstract view of interactions in terms of CSP events. We envisage that native code is only significant in implementing low-level routines for device interaction and hardware access, and these are fundamentally not modelled in detail, but instead captured by events. Where JNI may be used in other ways, for instance, to exploit performance-enhancing hardware, future extensions of our translation strategy are conceivable that incorporate models of JNI methods that have to be explicitly provided by the user.

The following classes are allowed as part of the **IC** category, but we do not provide formal models for them.

1. Classes related to interaction with devices and external events such as `InterruptServiceRoutine`.
2. Classes related to signals and happenings, such as `POSIXSignal` and `Happening`.

Since we take an abstract view of device interactions as CSP communications, we do not model the above classes. We require though that their design justifies the assumption that device access is atomic. In other words, the program should not be able to make any observations about the mechanisms that read (or write) data from (or to) the hardware, only that the data is made available as a whole. In particular, this has to be the case if complex data is communicated via inputs and outputs. Device access, however, does not have to be instantaneous, like CSP communications. Common modelling techniques in CSP can be used to explicitly model device interactions that take time, namely, as a pair of communications. Where such interactions can be regarded as virtually instantaneous, by which we mean they take a negligible amount of time, we encode them as single events. Although the implementation in that case weakens the precise timing behaviour, we stipulate that a similar argument as devised in [42] for timed automata can be construed to formally justify correctness of implementations in *Circus*, that is, under weaker notions of refinement such as ε-bisimulation.

To summarise, the SCJ developer has to guarantee that the program code for device access establishes at least atomicity of data being read or written. Virtual instantaneity is required only if device interactions are modelled by single events. The choice to adopt the single-event approach is, however, even then an issue of design that rests with the user of our technique.

We discuss a few basic architectural patterns for device interaction in Section 7.2, but in general such patterns are *per se* not a concern of the translation strategy. We also note that the language constraints in Section 6.1.2 do not apply to classes in the category **IC** and interaction code, for the reason that they are not subject to compositional translation.

Despite the restrictions presented in this section, it turned out to be possible to support our case study here as well as those in [36, 43] after refactoring. In the next section, we present our translation strategy applicable to all programs that fulfil the above constraints.

6.2. Translation Process

Translation is carried out by a three-stage process.

1. Analysis and annotation of the program code.
2. Automatic rewriting to transform program statements into a canonical form.
3. Translation of classes by category-specific rules.

Stage (1) is organised in four steps (1a) to (1d). We explain each of them next. Stages (2) and (3) are fully automatable via the rules we present later on.

Step (1a). In the first step, we proceed with an analysis of the program structure. We categorise all

Type	Category	Classes
active class	**SMMC**	`ACCSafelet`, `ACCMissionSequencer` and `ACCMission`
active class	**HC**	`WheelShaft`, `Engine`, `Brake`, `Gear`, `Lever`, `ThrottleController` and `SpeedMonitor`
data object	**DC**	`Controller`
interaction class	**IC**	`WheelShaftISR`, `EngineISR`, `BrakeISR`, `GearISR` and `LeverISR`

TABLE 5. Classification of all classes of the cruise controller.

classes of the SCJ program according to Table 4. To determine the active classes, namely those belonging to **SMMC** and **HC**, it is sufficient to examine the superclasses and implemented interfaces of a class. For the remaining classes, we have to make an intelligent decision whether they are data objects or interaction classes. Typically, interaction classes derive from classes of the 'Interaction with Devices and External Events' subclass hierarchy of the SCJ API [2]. There, we have classes such as `InterruptServiceRoutine` or `Happening`. Nevertheless, we do not categorically exclude non-SCJ classes being part of interaction patterns. Therefore, establishing membership of a class to either **DC** or **IC** in general requires insight and understanding of the program design and code, although there is scope for automation, too. In practical terms, the classification between **DC** or **IC** is encoded by custom Java annotations discussed in the next section. The annotations are an input for the translation in Stage (3).

Table 5 summarises the classification of the classes of the ACC. We observe that for each aperiodic handler (active class) we have a corresponding interrupt service routine (interaction class), suffixed with `ISR`, that releases (fires) the handler. In contrast, there is only one class `Controller` for a data object.

Step (1b). In the second step, we identify the instances (rather than classes) of missions and handlers of an application and introduce unique identifiers for each of them (of type *MissionId* and *HandlerId*). This constitutes a second input of the subsequent translation and again requires understanding of the SCJ program. As before, we encode this knowledge by way of special annotations in the program. For the ACC, we have a single identifier *ACCMId* of type *MissionId* for the single instance of the `ACCMission` class, and the identifiers *WheelShaftHId*, *EngineHId*, and so on, of type *HandlerId*, for the various handler instances.

Step (1c). In a third step, we determine the external events that cause the release of aperiodic handlers in the program, and annotate the respective handlers classes accordingly. This requires inspection of the mission initialisation methods that create the handlers.

Step (1d). In a final step, all methods of the **SMMC** and **HC** classes are annotated as to whether their bodies perform device interactions or external hardware accesses. For methods that do so, our analysis also has to provide a model for the device access as an action. Those methods are not translated by our strategy: their explicit action model is used instead.

Stage (1) is completed when the program is fully annotated to provide all necessary information to facilitate the (automatic) translation of the classes in Stage (3). It is worth noting that although Stage (1) in general requires human assistance, essentially no knowledge of formal models and *Circus* is needed for the steps (1a) to (1c). We furthermore stipulate that a tool could be developed to automate the annotation process, and, if not fully complete it, at least yield a good approximation. This is ongoing work.

Stage (2). The rewriting rules we apply to program statements in Stage (2) are summarised in Table 6. We use the function $[\![\ldots]\!]_{Rewrite}$ to capture the rewriting transformations. (Other functions $[\![\ldots]\!]\_$ with different subscripts are defined later on when formalising the translation.) To highlight meta-variables, we underline them as in $\underline{Expr}$ and $\underline{Meth}$. The name of the meta-variable also implicitly determines the syntactic category of the element matched. The abstract syntax we consider is the typical one of Java [44].

RW1 deals with local variables (as opposed to class fields) being initialised upon declaration. **RW2a** and **RW2b** rewrite the shorthand forms for the increment and decrement operators, which we support as stand-alone statements. **RW3** rewrites compound assignments into standard assignments. Lastly, **RW4** and **RW5** deal with truncated forms of the conditional and switch statements.

In the remainder of this section, we individually discuss the translation of active classes and data objects in Stage (3). As mentioned before, interaction classes are not modelled and thus not translated.

6.3. Active Classes

We have one translation rule for each type of active class, namely for the safelet, mission sequencer, a mission or a handler. The translation rule for a safelet, for example, is presented in Fig. 22. It matches all classes that implement the `Safelet` interface. Meta-variables of the rule, which as before are underlined, are $\underline{CName}$, $\underline{FieldDecls}$, $\underline{InitStmts}$, $\underline{AMethDecl1}$, $\underline{AMethDecl2}$, and so on.

The right-hand side of the rule constructs the process that models the class (that is, its active behaviour). We note that in this section, we only consider the process

Rule	Application	Result
RW1	$[\![$ *Type Var* = *E*; $]\!]_{Rewrite}$ where the declaration occurs in a statement block.	*Type Var*; *Var* = *E*;
RW2a	$[\![$ *Var* ++; $]\!]_{Rewrite}$	*Var* = *Var* + 1;
RW2b	$[\![$ *Var* --; $]\!]_{Rewrite}$	*Var* = *Var* - 1;
RW3	$[\![$ *Var Op* = *E*; $]\!]_{Rewrite}$	*Var* = *Var Op E*;
RW4	$[\![$ if (*Expr*) *Stmts* $]\!]_{Rewrite}$	if (*Expr*) *Stmts* else { }

RW5	$[\![$ switch (*Expr*) {
	case $Value_1$: $Stmts_1$; break;
	case $Value_2$: $Stmts_2$; break;
	...
	case $Value_n$: $Stmts_n$; break;
	} $]\!]_{Rewrite}$
		switch (*Expr*) {
		case $Value_1$: $Stmts_1$; break;
		case $Value_2$: $Stmts_2$; break;
		...
		case $Value_n$: $Stmts_n$; break;
		default: { }
		}

TABLE 6. Rewrite rules for transformation into canonical statements.

$$
[\![\text{class } \underline{CName} \text{ implements Safelet } \{ \\
\quad \underline{FieldDecls} \\
\quad \text{public } \underline{CName}() \{ \underline{InitStmts} \} \\
\quad \underline{AMethDecl1} \\
\quad \underline{AMethDecl2} \\
\quad \ldots \\
\quad \text{public MissionSequencer getSequencer() } \{ \\
\quad\quad \text{return new } \underline{MSName}(); \\
\quad \} \\
\}]\!] \;\widehat{=}
$$

$$
\begin{array}{l}
\textbf{process } [\![\underline{CName}]\!]_{Name} App \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state } State == [this : [\![\underline{CName}]\!]_{Name} Class] \\
\quad Init == [State' \mid this' = \textbf{new } [\![\underline{CName}]\!]_{Name} Class] \\
\quad [\![\underline{AMethDecl1}]\!]_{AMethod} \\
\quad [\![\underline{AMethDecl2}]\!]_{AMethod} \\
\quad \ldots \\
\quad Methods \mathrel{\widehat{=}} \\
\qquad \left(\mu X \bullet \left(\begin{array}{l} [\![\underline{AMethDecl1}]\!]_{Name} Meth \;\Box \\ [\![\underline{AMethDecl2}]\!]_{Name} Meth \;\Box \\ \ldots \end{array} \right) ; \; X \right) \\
\quad \bullet Init \; ; \; (Methods \triangle end\_safelet\_app \longrightarrow \textbf{skip}) \\
\textbf{end}
\end{array}
$$

FIGURE 22. Translation rule for classes implementing the `Safelet` interface.

model of active components. The associated *OhCircus* class, capturing state and data operations, is generated by the same translation rule that we use for passive classes and discuss in Section 6.5.

The rule makes use of two auxiliary translation functions, $[\![\ldots]\!]_{Name}$ for translating identifier names and $[\![\ldots]\!]_{AMethod}$ for translating methods into actions. Suffixes in the translation functions are used for clarity and also to avoid ambiguities later on when presenting complementary rules for passive classes. The function $[\![\ldots]\!]_{Name}$ is used to encode an identifier for a class, variable, or method into a valid Z identifier. For classes, this uses the fully-qualified name of the class and replaces '.' separators by underscores. Other symbols not permitted in *Circus* are substituted suitably as well. The function furthermore ensures that method names are unique in the presence of method overloading. The function $[\![\ldots]\!]_{AMethod}$ encodes a class method as a local action definition. It is applied to all methods of the class that are modelled by actions. Such methods are, first of all, the `setUp()` and `tearDown()` methods, but also possibly other methods that interact with devices and hardware and thus require action models.

Although not made explicit in the rules, rule matching exploits the presence of annotations that indicate whether a method is to be translated into an action or a data operation. We shall not specify the details of the annotation mechanism here as this is rather a technical issue for a tool and discussed in the next section. Instead, we implicitly assume that *AMethDecl1*, *AMethDecl2*, and so on, only match methods that are annotated to require action models. Other methods are simply ignored by the matching process and become significant only in constructing the class model. The *FieldDecls* meta-variable is matched but, not used by the rule in Fig. 22. It is instantiated with the declaration of instance variables of the class.

In what follows, we individually discuss the elements of the process resulting from the translation, that is its *State*, *Init* action, method actions, and main action.

Process State. The translation constructs the state paragraph of the process using a schema *State*, which includes a single component *this* whose type

$$\left[\!\!\left[\begin{array}{l} \underline{AccessModifier}\ \texttt{synchronized} \\ \underline{RetType}\ \underline{MethName}(\underline{Args})\ \{\ \underline{Body}\ \} \end{array} \right]\!\!\right]_{Method} \mathrel{\widehat=} \boxed{\begin{array}{l} [\![\underline{MethName}]\!]_{Name} Meth \mathrel{\widehat=} \\ [\![\underline{MethName}]\!]_{Name} Call\,?\,args \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{var}\ ret : [\![\underline{RetType}]\!]_{Type} \bullet [\![\underline{Body}]\!]_{Stmts}; \\ [\![\underline{MethName}]\!]_{Name} Ret\,!\,ret \longrightarrow \mathbf{skip} \end{array} \right) \end{array}}$$

FIGURE 23. Translation rule for an interacting method into a local action.

$$\left[\!\!\left[\begin{array}{l} \underline{AccessModifier}\ \texttt{synchronized} \\ \texttt{void}\ \underline{MethName}()\ \{\ \underline{Body}\ \} \end{array} \right]\!\!\right]_{Method} \mathrel{\widehat=} \boxed{\begin{array}{l} [\![\underline{MethName}]\!]_{Name} Meth \mathrel{\widehat=} \\ \quad [\![\underline{MethName}]\!]_{Name} Call \longrightarrow [\![\underline{Body}]\!]_{Stmts}; \\ \quad [\![\underline{MethName}]\!]_{Name} Ret \longrightarrow \mathbf{skip} \end{array}}$$

FIGURE 24. Alternative rule for an interacting method without arguments and a return value.

is that of the underlying data object. Whereas the process name is defined by $[\![\underline{CName}]\!]_{Name} App$, the name of the respective *OhCircus* class is defined by $[\![\underline{CName}]\!]_{Name} Class$. This realises the wrapping pattern described in Fig. 13. The result of the translation for our example safelet is exactly the process in Fig. 14 after a trivial simplification to remove the empty state paragraph and initialisation action.

Init Action. The initialisation action assigns a new instance of a class object to *this*. We note that the creation of the class object also entails the constructor being executed, so that the fields of the class object are suitably initialised. The constructor itself, whose body is given by *InitStmts*, is modelled in the corresponding *OhCircus* class rather than in the process in Fig. 22.

Method Actions. Interacting and infrastructure methods of an active class are translated, as already mentioned, by $[\![\ldots]\!]_{AMethod}$. Each method results in the declaration of a local action with suffix *Meth*. The corresponding rule is in Fig. 23. The rule makes explicit that we require all methods of active classes to be synchronised. The rule is applicable to methods that have both a return value and parameters. The name of the local action is derived from the name of the method in the program and so are the names of the *Call* and *Ret* channels on which we synchronise to call the method and to wait for its completion. (We also have translation rules that introduce those channel pairs.)

We use two further translation functions in the safelet rule: $[\![\ldots]\!]_{Type}$ and $[\![\ldots]\!]_{Stmts}$. The $[\![\ldots]\!]_{Type}$ function encodes a Java type as a corresponding Z (and, therefore, *Circus*) type. Its definition is included in Appendix D.1. We support all primitive types of Java, although we do not define a model for `float` and `double` at present. Future work may address this, using, for instance, an axiomatisation of the real numbers in Z like the one in [45]. Z types are introduced to represent the various semantic domains for primitive Java types. Reference types refer to *OhCircus* classes that result from the translation of data objects and have a suffix *Class*. Arrays are modelled by sequences and we provide generic functions to construct array types and to get and set the elements of an array.

The $[\![\ldots]\!]_{Stmts}$ function translates a statement block (here the method body) into an action term; we discuss it in more detail in the remainder of this section. The bound variables *args* and *ret* are visible by the translation of the body, and can be used to refer to the arguments of a call and to set the return value.

Returning from a method results in assigning the return value to *ret*. This is captured by the following translation rule for a `return` statement.

$$[\![\texttt{return}\ \underline{Expr}]\!]_{Stmts} \mathrel{\widehat=} ret := [\![\underline{Expr}]\!]_{Expr}$$

The translation function $[\![\ldots]\!]_{Expr}$ translates a (side-effect free) expression; we discuss it in the sequel. The suitability of the rule above, as well as the one in Fig. 23 is contingent on no statement(s) following a `return`. This is, however, ensured by our structural constraints.

For methods that do not have parameters or a `void` return type, we provide rules that take a simpler shape. There, the input and output prefixes become simple synchronisations as illustrated in Fig. 24.

Table 7 presents the most interesting rules for statement translation. They cater for assignments (**SR1**), local variables (**SR2**), conditional statements (**SR3**), switch statements (**SR4**), while-loops (**SR5**) and assertions (**SR6**). The remaining rules can be found in Table D.5 of Appendix D. They deal with statement sequences (**SR7**–**SR8**), blocks (**SR9**) and for-loops (**SR10**) and are mostly straightforward.

We point out that in **SR2**, *Stmts* includes all remaining statements of the current block. Also, in the conditional of the right-hand action of **SR4**, no nondeterminism can arise since compilation enforces the values $\underline{Value}_1$, $\underline{Value}_2$, and so on, to be distinct. While-loops (**SR5**) are modelled by a recursion. Depending on the value of the loop condition, we either execute the loop body and recurse, or terminate the loop via a **skip** action. The absence of `break` and `continue` statements ensure that this is a correct model.

As an example, we give the translation of part of the `handleAsyncEvent()` method in the `ThrottleController` class. An extract of this method, after rewriting, is included in Fig. 25. The translation uses the rule **SR3** to translate the `if` statements, **SR1** and **SR2** to translate the assignment and variable declaration, and **SR10** and **SR11** (Appendix D) to

Rule	Application	Result
SR1	$[\![\underline{Var} \texttt{ = } \underline{Expr} \texttt{;}]\!]_{Stmts}$	$[\![\underline{Var}]\!]_{Name} := [\![\underline{Expr}]\!]_{Expr}$
SR2	$[\![\underline{Type}\ \underline{Var} \texttt{;}\ \underline{Stmts}]\!]_{Stmts}$	$\mathbf{var}\ [\![\underline{Var}]\!]_{Name} : [\![\underline{Type}]\!]_{Type} \bullet [\![\underline{Stmts}]\!]_{Stmts}$
SR3	$[\![\texttt{if (} \underline{Expr} \texttt{)}\ \underline{Stmts_1}\ \texttt{else}\ \underline{Stmts_2}]\!]_{Stmts}$	$\mathbf{if}\ \ [\![\underline{Expr}]\!]_{Expr} \longrightarrow [\![\underline{Stmts_1}]\!]_{Stmts}$ $[]\ \neg\ [\![\underline{Expr}]\!]_{Expr} \longrightarrow [\![\underline{Stmts_2}]\!]_{Stmts}$ $\mathbf{fi}$
SR4	$\left[\!\!\left[\begin{array}{l} \texttt{switch (}\underline{Expr}\texttt{) \{} \\ \quad \texttt{case}\ \underline{Value_1}\texttt{:}\ \underline{Stmts_1}\texttt{; break;} \\ \quad \texttt{case}\ \underline{Value_2}\texttt{:}\ \underline{Stmts_2}\texttt{; break;} \\ \quad \ldots \\ \quad \texttt{default:}\ \underline{Stmts} \\ \texttt{\}} \end{array} \right]\!\!\right]^{\sigma}_{Stmts}$	$\mathbf{if}\ [\![\underline{Expr}]\!]_{Expr} = [\![\underline{Value_1}]\!]_{Value} \longrightarrow [\![\underline{Stmts_1}]\!]_{Stmts}$ $[]\ [\![\underline{Expr}]\!]_{Expr} = [\![\underline{Value_2}]\!]_{Value} \longrightarrow [\![\underline{Stmts_2}]\!]_{Stmts}$ $[]\ \ldots$ $[]\ \neg \left(\begin{array}{l} [\![\underline{Expr}]\!]_{Expr} = [\![\underline{Value_1}]\!]_{Value} \lor \\ [\![\underline{Expr}]\!]_{Expr} = [\![\underline{Value_2}]\!]_{Value} \lor \ldots \end{array} \right) \longrightarrow [\![\underline{Stmts}]\!]_{Stmts}$ $\mathbf{fi}$
SR5	$[\![\texttt{while (}\underline{Expr}\texttt{)}\ \underline{Stmts}]\!]_{Stmts}$	$\mu X \bullet \left(\begin{array}{l} \mathbf{if}\ [\![\underline{Expr}]\!]_{Expr} \longrightarrow [\![\underline{Stmts}]\!]_{Stmts}\ ;\ X \\ []\ \neg\ [\![\underline{Expr}]\!]_{Expr} \longrightarrow \mathbf{skip} \\ \mathbf{fi} \end{array} \right)$
SR6	$[\![\texttt{assert}\ \underline{Expr}\texttt{;}]\!]_{Stmts}$	$\mathbf{if}\ [\![\underline{Expr}]\!]_{Expr} \longrightarrow \mathbf{skip}\ []\ \neg\ [\![\underline{Expr}]\!]_{Expr} \longrightarrow \mathbf{abort}\ \mathbf{fi}$

TABLE 7. Translation rules for sequential statements.

translate statement sequences and blocks. The result of the translation is given by the following action.

$$
\begin{array}{l}
\mathit{handleAsyncEventMeth} \mathrel{\widehat=} \\
\quad \mathbf{if}\ \mathit{this.schedule\_throttle} = \mathit{TRUE} \longrightarrow \\
\qquad \mathbf{if}\ \mathit{this.accelerating} = \mathit{TRUE} \longrightarrow \\
\qquad\quad \mathit{increaseVoltageCall} \longrightarrow \mathbf{skip}; \\
\qquad\quad \mathit{increaseVoltageRet} \longrightarrow \mathbf{skip} \\
\qquad [] \mathit{this.accelerating} = \mathit{FALSE} \longrightarrow \\
\qquad\quad \mathbf{if}\ \mathit{this.maintainSpeed} = \mathit{TRUE} \longrightarrow \\
\qquad\qquad \mathbf{var}\ \mathit{current\_speed} : \mathit{int} \bullet \\
\qquad\qquad \mathit{current\_speed} := \mathit{this.speedo.getCurrentSpeed}(); \\
\qquad\qquad \mathbf{if}\ \mathit{this.cruiseSpeed} - \mathit{current\_speed} > 2 \longrightarrow \\
\qquad\qquad\quad \mathit{increaseVoltageCall} \longrightarrow \mathbf{skip}; \\
\qquad\qquad\quad \mathit{increaseVoltageRet} \longrightarrow \mathbf{skip} \\
\qquad\qquad [] \mathit{this.cruiseSpeed} - \mathit{current\_speed} < -2 \longrightarrow \\
\qquad\qquad\quad \mathit{resetVoltageCall} \longrightarrow \mathbf{skip}; \\
\qquad\qquad\quad \mathit{resetVoltageRet} \longrightarrow \mathbf{skip} \\
\qquad\qquad [] \neg (\ldots) \longrightarrow \ldots \\
\qquad\qquad \mathbf{fi} \\
\qquad\quad [] \mathit{this.maintainSpeed} = \mathit{FALSE} \longrightarrow \mathbf{skip} \\
\qquad\quad \mathbf{fi} \\
\quad \mathbf{fi} \\
\quad [] \mathit{this.schedule\_throttle} = \mathit{FALSE} \longrightarrow \mathbf{skip} \\
\quad \mathbf{fi}
\end{array}
$$

References to instance variables are via the aggregated class object *this*. The dots correspond to the translation of the code that has been omitted in Fig. 25. To translate calls to the methods `increaseVoltage()` and `resetVoltage()`, we require further rules. These are given in Table D.6 in Appendix D. The semantics varies depending on (a) the type of the target object and (b) whether the invocation assigns the return value to a variable. In the example above, we use the rule for a method call in an assignment where the target is the safelet or mission sequencer (**SR11**). Since there is only one safelet and mission sequencer in a valid SCJ program, the corresponding channels are not parametrised by a type-specific identifier.

For missions and handlers (**SR13** and **SR14**), we require an additional channel parameter to identify the target object, as there potentially may be more than one, namely if we have multiple missions and handlers. The association of classes with identifiers is determined, as already explained, by annotations which are created during the analysis and configuration in Stage (1).

Finally, to define the behaviour of the application process, we generate an action *Methods* (Fig. 22) that offers to other processes the choice of calling one of the interacting methods of the class.

Main Action. The main action is completely generic and always has the same shape, firstly executing the initialisation action *Init*, and then offering a call to one of the methods until the process is terminated. Here, we take advantage of the structural constraint that all methods of active classes are `synchronized`. In cases where the class does not have instance variables, like the `ACCSafelet` class of the ACC, simplifications are possible. They are obvious and thus their translation rules are not further discussed here.

In addition to a process for each active class, we moreover have to introduce declarations for the method-channel pairs of non-infrastructure methods of the class, since those are application specific. The corresponding rules are applied independently of the rules for translating classes into processes and give rise to separate specification paragraphs in the generated models. These rules are not very interesting though, therefore a detailed discussion is omitted.

```
public void handleAsyncEvent() {
  if (schedule_throttle) {
    if (accelerating) {
      increaseVoltage();
    }
    else {
      if (maintainSpeed) {
        int current_speed;
        current_speed = speedo.getCurrentSpeed();
        if (cruiseSpeed - current_speed > 2) {
          increaseVoltage();
        }
        else if (cruiseSpeed - current_speed < -2) {
          resetVoltage();
        }
        else { /* Remainder of the code is omitted! */ }
      }
    }
  }
}
```

FIGURE 25. `handleAsyncEvent()` method of the `ThrottleController` class.

The translation rules for the mission sequencer and missions are included in Appendix C; they are very similar to the safelet rule just discussed. We have a minor deviation in the main action of the translation rule for a mission that ensures that the mission application process is restarted after termination of the mission so that the mission can be executed more than once. The rule for handlers is more interesting: we discuss it in more detail in the next section.

6.4. Handler Classes

Handler classes are essentially active classes. Their model slightly differs though in terms of the process structure. The translation rule for an aperiodic handler is included in Fig. 26. It defines the translation of handler classes that either extend `AperiodicEventHandler` or `AperiodicLongEventHandler`.

Unlike in the process models for the safelet and the mission sequencer, the coupling between process and data object in handler processes is dynamic. Hence, we introduce a channel communication $[\![\underline{CName}]\!]_{Name} Init$ upon construction of a handler instance (see Fig. 26). The handler process synchronises on this channel in the *Init* action to establish the link to the underlying *OhCircus* class object that captures the state and data operations of the SCJ class. In the model of the ACC, this is illustrated by the handler processes in Fig. 19 and Fig. 20. The initialisation channels these handlers synchronise on are *EngineInit* and *ThrottleControllerInit*. At the point of synchronisation, the *OhCircus* class objects have already been created with their fields initialised by a class constructor.

The execution behaviour of handlers (defined by *Execute*) is specified in a way that slightly differs from the pattern for classes belonging to **SMMC**. This is because we require additional active behaviour in order to release the handler, in addition to offering calls to interacting methods. As already said, *Dispatch* takes care of both method calls and handler releases. It also enables termination of the handler via a handler-specific *leave_dispatch . h* event, parametrised by a handler identifier h. Execution starts with a synchronisation on *enter_dispatch .h*.

The function BoundEvents in Fig. 26 determines the bound events of a handler class. For each handler class of an application, we specify the bound events as part of the analysis and configuration in Stage (1c).

When an external bound event occurs, the action *handleAsyncEvent*, encoding the handler method, is invoked. Its behaviour is determined by the body $\underline{HdlStmts}$ of the handler method. We use an auxiliary translation function $[\![\_]\!]_{HdlBody}$ which is defined by

$$[\![\underline{HdlStmts}]\!]_{HdlBody} \mathrel{\widehat{=}}$$

$$\boxed{\begin{array}{l}\textbf{wait } [\![\underline{CName}]\!]_{Name}\_handleAsyncEvent_{TB}; \\ this.handleAsyncEvent(inp)\end{array}}$$

if the handler method is a data operation, and by

$$[\![\underline{HdlStmts}]\!]_{HdlBody} \mathrel{\widehat{=}} [\![\underline{HdlStmts}]\!]_{Stmts}$$

if the handler method interacts with external devices or hardware. In the non-interacting case, translation uses the encoding of the handler method as an *OhCircus* method in the underlying *OhCircus* class. The constant that is obtained by suffixing the class name with $handleAsyncEvent_{TB}$ specifies the time budget of the handler. Otherwise, the method is given an action model using $[\![\_]\!]_{Stmts}$ like other interacting methods.

The application process for a periodic event handler requires a modification in the definition of *Dispatch* and

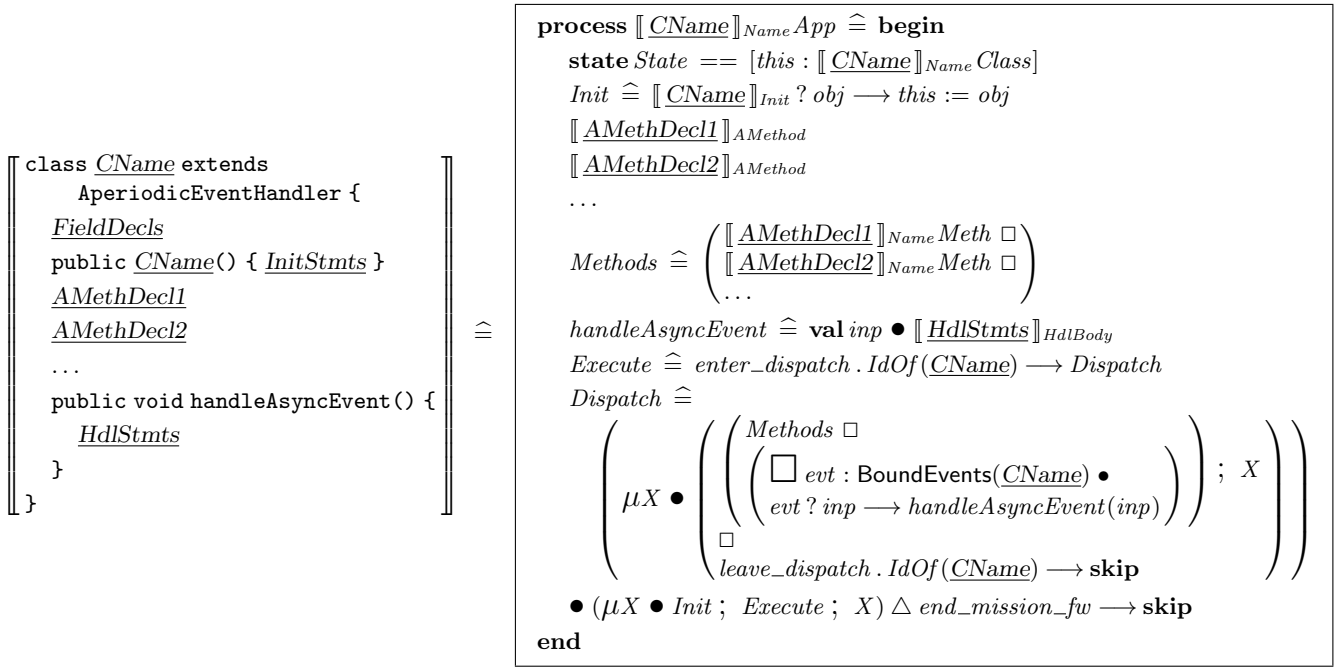

FIGURE 26. Translation rule for classes extending the `AperiodicEventHandler` class.

a supplementary action *Release*. The corresponding rule is included in Appendix C (Fig. C.5). There, the handler is released by an internal handler-specific timer event *release_handler . IdOf*(CName) rather than an external event, as it is the case for aperiodic handlers.

Having explained in detail how active classes are given a process model, in the next subsection we examine the translation of data objects.

6.5. Data Objects

We next define the translation of classes belonging to the category **DC**. They do not interact with the SCJ infrastructure or external devices. The high-level translation rule for such classes is presented in Fig. 27. The result of the translation is an *OhCircus* class. In the rule, we assume that the class extends another class CBase and explicitly calls one of its super constructors. An alternative rule also exists for the simpler case where we do not have inheritance. We omit its detailed specification, which can be obtained by removing the '`extends`' clauses in Fig. 27. We recall that active classes, namely those in category **SMMC** and **HC**, are also translated using the rules for data objects since they give rise to both a process and an *OhCircus* class (see Table 5.1). In active classes, however, only the non-interacting methods are considered and neither do we consider methods called by other active components.

The name of the *OhCircus* class is derived from the name of the of Java class, suffixed with *Class*. As in the rule for active classes, we first have a **state** paragraph that determines the state of a class object. The state paragraph is constructed from the declaration of non-static instance variables of the class. The translation functions $[\![\ldots]\!]_{Fields}$ and $[\![\ldots]\!]_{FieldInit}$ apply to (lists of) field declarations and are used to generate the state components and the initialisation schema of the class. Field initialisation takes place upon declaring the variables and is captured by the $[\![\underline{CName}]\!]_{Name} Init$ schema operation.

For $[\![\ldots]\!]_{Fields}$ we have the rules in Table D.4 in Appendix D. They make use of the functions $[\![\ldots]\!]_{Name}$ and $[\![\ldots]\!]_{Type}$ that have already been mentioned, and are specified in Appendix D, too. For field initialisation, we have the rules in Table 8. There, $[\![\underline{FieldType}]\!]_{DefaultInit}$ is defined to yield the default value of a primitive type. We omit its definition, which is in line with the Java language specification [44].

AccessModifier stands for any of the access modifiers `public`, `protected` or `private`. The translation of access modifiers is trivial as it merely replaces them by the corresponding *OhCircus* keywords: **public**, **protected** and **private**. These keywords enforce similar visibility restrictions as in the Java language. Notably, the **private** keyword reflects the particular class-based semantics of private access adopted by Java. We point out that we do not capture package access in our model. Chiefly, because *OhCircus* does not have a structuring notion equivalent to Java packages.

We recall that the function $[\![\ldots]\!]_{Expr}$ translates an expression. An extract of its definition is in Table D.3 in Appendix D. This is done in the usual manner, encoding Java operators using corresponding Z operators on primitive types. Because of the restrictions in Section 6.1.2, and prior rewriting, an expression cannot have side effects unless it is the right-hand of an assignment. Therefore, all method calls that occur in

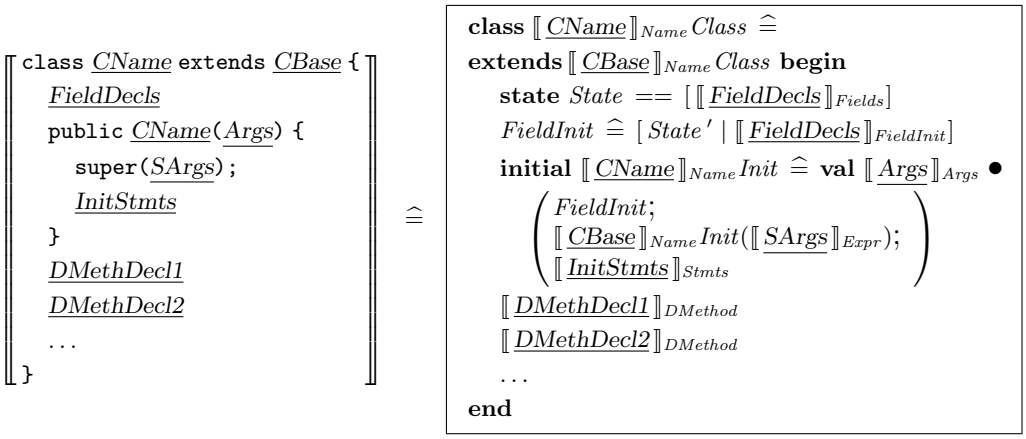

FIGURE 27. Translation rule for a data object class.

Rule	Application	Result
IR1	$\llbracket \underline{FieldDecl} \; \underline{FieldDecls} \rrbracket_{FieldInit}$ where *FieldDecls* is a non-empty list of field declarations	$\llbracket \underline{FieldDecl} \rrbracket_{FieldInit} \wedge \llbracket \underline{FieldDecls} \rrbracket_{FieldInit}$
IR2	$\llbracket \underline{AccessModifier}\ \underline{FieldType}\ \underline{FieldName}\ \texttt{;} \rrbracket_{FieldInit}$	$\llbracket \underline{FieldName} \rrbracket_{Name}{}' = \llbracket \underline{FieldType} \rrbracket_{DefaultInit}$
IR3	$\llbracket \underline{AccessModifier}\ \underline{FieldType}\ \underline{FieldName}\ \texttt{=}\ \underline{Expr}\ \texttt{;} \rrbracket_{FieldInit}$	$\llbracket \underline{FieldName} \rrbracket_{Name}{}' = \llbracket \underline{Expr} \rrbracket_{Expr}$

TABLE 8. Translation rules for field initialisations.

an expression can be modelled by *OhCircus* **function** invocations. The corresponding rule for translating method calls in expressions is as follows.

$$\llbracket \underline{Obj}.\underline{Meth}(\underline{Args}) \rrbracket_{Expr} \mathrel{\widehat=} \llbracket \underline{Obj} \rrbracket_{Expr} \cdot \llbracket \underline{Meth} \rrbracket_{Name}(\llbracket \underline{Args} \rrbracket_{Expr})$$

It only applies to methods that are data operations (non-interacting). Method calls that do provoke side effects can only occur in the restricted forms `Obj.Meth(Args);` and `Var = Obj.Meth(Args);` and a specific rule is applied to the latter that introduces the assigned value as a result parameter of the method.

The **initial** paragraph encodes the construction of the class object. This involves field initialisation, the invocation of the superclass constructor, and the statements of the constructor. As we support constructors with arguments, we have to account for them in the translation. The auxiliary $\llbracket \ldots \rrbracket_{Args}$ function yields the parameter declarations corresponding to the call signature of a constructor or method. If applied to an argument tuple, $\llbracket \ldots \rrbracket_{Args}$ distributes through that tuple. If applied to a single argument, we have the rule below.

$$\llbracket \underline{Type}\ \underline{Var} \rrbracket_{Args} \mathrel{\widehat=} \llbracket \underline{Var} \rrbracket_{Name} : \llbracket \underline{Type} \rrbracket_{Type}$$

For example, **val** $\llbracket$`(String name, int age)`$\rrbracket_{Args}$ evaluates to **val** *name* : *StringClass*; *age* : *int*.

The constructor, whose body is given by *InitStmts*, is translated into an *OhCircus* method. The function $\llbracket \ldots \rrbracket_{Stmts}$ is used again for this purpose. As before, it translates a Java statement block. Here, however, the result is the description of an *OhCircus* method rather than an action. The rules in Table 7 nevertheless still apply; for the translation of method calls (Appendix D.6) we have a different rule that translates them into *OhCircus* method calls rather than synchronisations on *Call* and *Ret* channels.

In translating method declarations, we introduce the function $\llbracket \ldots \rrbracket_{DMethod}$ rather than $\llbracket \ldots \rrbracket_{AMethod}$ since there is a fundamental difference with respect to active classes. It encodes a (non-static) class method as an *OhCircus* method rather than an action.

$$\llbracket \underline{AccessModifier}\ \underline{Type}\ \underline{Meth}(\underline{Args})\ \{\ \underline{Body}\ \} \rrbracket_{DMethod} \mathrel{\widehat=} \textbf{val}\ \llbracket \underline{Args} \rrbracket_{Args};\ \textbf{res}\ ret : \llbracket \underline{Type} \rrbracket_{Type} \bullet \llbracket \underline{Body} \rrbracket_{Stmts}$$

The main difference is that we do not have channel communications as in Fig. 23. Instead, value and result parameters are used for passing arguments and setting the return value. The method body is translated by $\llbracket \ldots \rrbracket_{Stmts}$ (defined in Table 7). The rules for object creation and method calls in Tables D.6 and D.7, however, do not apply. Instead, we use the **new** and method call constructs provided by *OhCircus*.

To conclude our account on the translation strategy, we recall that certain methods are excluded from the application of the translation rules. Namely, these are the methods that carry out device or hardware accesses. Here, annotations explicitly provide action models for the methods, and these models are directly used in the processes rather than the result of $\llbracket \ldots \rrbracket_{AMethod}$.

In the next section we discuss implementation issues and examine how translation of the ACC program is

Java Annotation	Parameters	Determines
`@ActiveData`	none	classes in category **SMMC** or **HC**
`@PassiveData`	none	classes in category **DC**
`@InteractionClass`	none	classes in category **IC**
`@MissionId(id:String)`	`id` – mission identifier	identifier of a mission class
`@HandlerId(id:String)`	`id` – handler identifier	identifier of a handler class
`@BoundEvent(channel:String, type:String)`	`channel` – name of the channel `type` – type of channel if present	external event bound to a handler
`@BoundEvents`	`BoundEvent[] events` – set of events	multiple events bound to a handler
`@DeviceAccess(model:String)`	`model` – *Circus* model of the access	action that models a device access
`@InteractionCode(model:String)`	`model` – *Circus* model if applicable The default value for `model` is **skip**.	fields and methods for interaction code
`@Ignore`	none	explicitly ignored program elements

TABLE 9. Java annotations to configure the translation of SCJ programs.

performed automatically by our tool.

7. AUTOMATIC TRANSLATION

To automate the translation of SCJ programs into *Circus* models according to the rules presented in the previous section, we have developed a translation tool. It takes a translatable program according to the restrictions in Section 6.1 as an input and produces a collection of *Circus* specification files, encoded in the LaTeX mark-up format understood by the *Circus* parser of the Community Z Tools (CZT) [34]. Our tool is freely available from the hiJaC project page: `http://www.cs.york.ac.uk/circus/hijac/tools.html`.

The translator tool is implemented in Java and takes advantage of a number of third-party utilities and libraries. They are specifically the Compiler API of the JDK 7 [46], JSR 308, which extends the capabilities of Java's annotation mechanism [47], and the FreeMarker template engine [48]. Our tool design facilitates modifications and extensions of the translator and defines a clean component framework in which the core translator is traceable to our rules.

The configuration of the translation in Stage (1), as already explained, is realised by a set of custom Java annotations. They are summarised in Table 9. We discuss their individual purpose in the next section.

7.1. Annotation Framework

We can categorise the annotations into four groups.

1. Annotations that determine the categories of Java classes. In this group, we have `@ActiveData` for both **SMMC** and **HC**, `@PassiveData` for **DC** and `@InteractionClass` for **IC**.
2. Annotations that specify identifiers of SCJ classes in the *Circus* model. Here, we have `@MissionId` and `@HandlerId`. These annotations apply to classes only and have to be provided for all mission and handler classes of an SCJ application.
3. Annotations that associate external events with the handlers that are released by them. These are `@BoundEvent` and `@BoundEvents`.
4. Annotations that specify action models for device access and hardware interaction code. These are `@DeviceAccess` and `@InteractionCode`.

In addition, we included an extra annotation `@Ignore` above to tag elements of the program that ought to be explicitly ignored during model construction. We apply it, for instance, to method parameters of type `AperiodicEvent`, which we do not model, or calls to `super` constructors of handlers. We observe that Java does not allow multiple annotations of the same type on an element, which is why we require `@BoundEvents` in addition to `@BoundEvent`. The former simply aggregates annotations of the latter type.

There are no annotations in group (1) to determine membership of classes to the categories **SMMC** and **HC** *per se* since this can be easily determined by examining the class types. Classes that are annotated by `@InteractionClass` are ignored by the translator.

We point out that our annotations at present assign handler and mission identifiers to classes rather than their instances. For the examples we consider, this is sufficient. The integration of instance-based identifiers is work in progress and complicated by the fact that Java does not permit the annotation of individual statements such as the construction of an object.

In group (4), the `@DeviceAccess` annotation allows us to define an action model for an explicit device or hardware access. For instance, in the cruise controller, we have the function `writeVoltage()` in the `ThrottleController` class, which communicates on the *set_voltage* channel to output a voltage value to the throttle. The model of the device access is provided to the annotation as a `String`, by the `model` parameter. It usually is a prefix with possibly additional timing constraints attached to it. The annotation enables us to abstract from the low-level mechanics of the

```
@ActiveData
@MissionId("ACCMId")
class ACCMission extends Mission {

  /* Aperiodic Events */
  @InteractionCode private AperiodicEvent shaft_event;
  @InteractionCode private AperiodicLongEvent engine_event;
  @InteractionCode private AperiodicLongEvent brake_event;
  @InteractionCode private AperiodicLongEvent gear_event;
  @InteractionCode private AperiodicLongEvent lever_event;

  /* Interrupt Service Routines */
  @InteractionCode private WheelShaftISR shaft_isr;
  @InteractionCode private EngineISR engine_isr;
  @InteractionCode private BrakeISR brake_isr;
  @InteractionCode private GearISR gear_isr;
  @InteractionCode private LeverISR lever_isr;

  @InteractionCode
  private void createEvents() {
    shaft_event = new AperiodicEvent();
    engine_event = new AperiodicLongEvent();
    /* ... */
  }

  @InteractionCode
  private void createISRs() {
    shaft_isr = new WheelShaftISR(shaft_event);
    engine_isr = new EngineISR(engine_event);
    /* ... */
  }

  @InteractionCode
  private void registerISRs() {
    shaft_isr.register();
    engine_isr.register();
    /* ... */
  }

  @InteractionCode
  private void unregisterISRs() {
    shaft_isr.unregister();
    engine_isr.unregister();
    /* ... */
  }

  public void initialize() {
    createEvents();
    createISRs();
    registerISRs();
    /* Create event handlers and data objects. */
    WheelShaft shaft = new WheelShaft(shaft_event);
    SpeedMonitor speedo = new SpeedMonitor(shaft, 500);
    ThrottleController throttle = new ThrottleController(speedo);
    Controller cruise = new Controller(throttle, speedo);
    Engine engine = new Engine(cruise, engine_event);
    Brake brake = new Brake(cruise, brake_event);
    Gear gear = new Gear(cruise, gear_event);
    Lever lever = new Lever(cruise, lever_event);
    /* Register event handlers with the mission. */
    shaft.register();
    engine.register();
    brake.register();
    gear.register();
    lever.register();
    speedo.register();
    throttle.register();
  }

  public void cleanup() {
    unregisterISRs();
  }

  public long missionMemorySize() {
    return 131072;
  }
}
```

FIGURE 28. Annotated version of the `ACCMission` class of the cruise controller.

code as it encapsulates assumptions we make about the nature of interactions. The obligation to justify these assumptions rests, as previously noted, with the engineer writing the annotation and this is one of the few tasks that require insight and understanding of the execution environment. We note that the annotation `@InteractionCode`, like `@DeviceAccess`, permits the optional specification of a model for the code, too. This is for generality as one may envisage situations where hardware configuration code performs initialisation or finalisation steps that correspond to actual communications with the external environment; in our example, this is not the case though.

The `@InteractionCode` annotation enables us to make a finer distinction in tagging parts of the program that correspond to code that is needed to configure interrupts and hardware devices and thus usually ignored. It applies to methods and fields and typically identifies methods that create and register interrupt service routines, either for the entire application, or alternatively, single missions. Since standard Java does not support the annotation of individual program statements, we generally require that such interaction code is encapsulated cleanly into methods. This can always be achieved by refactoring, and the annotation stage normally reveals whether there is a need for it.

As suggested earlier on, it should be feasible to automate a significant portion of the annotation process. For instance, certain SCJ classes could by default be annotated as active objects, like those derived from `Safelet`, `MissionSequencer`, `Mission`, and so on. Other classes may similarly be automatically annotated as interaction classes, like those deriving from `InterruptServiceRoutine`. The identifiers for missions and handlers could, in principle, also be derived automatically. We observe, however, that the process could be challenging to automate as a whole due to the fact that human insight and understanding of the technology is required to specify the models for device interaction and to testify that the assumptions of atomic and instantaneous interactions are met.

In the next section we illustrate the use of the annotations presented above in the ACC.

7.2. Example: the Cruise Controller

In the ACC program, the classes `ACCSafelet` and `ACCMissionSequencer` require only `@ActiveData` annotations. This is because they have very simple implementations that do not include any code related to interaction with hardware or configuration of interrupt service routines. More interesting is the class `ACCMission`, which is presented in Fig. 28. Apart from the `initialize()` and `cleanup()` methods called by the infrastructure to initialise and finalise the mission, several additional methods deal with the creation and registration of interrupt service routines (ISRs). The ISRs are held by the instance variables `shaft_isr`,

```
@InteractionClass
public class EngineISR extends InterruptServiceRoutine {
  protected final AperiodicLongEvent engine_event;

  public EngineISR(AperiodicLongEvent event) {
    super("EngineISR");
    engine_event = event;
  }

  public void handle() {
    disableInterrupts();
    /* Determine external event that raised the interrupt. */
    long cause = ... ;
    engine_event.fire(cause);
    /* Interrupts are re-enabled by the aperiodic handler. */
  }

  public void disableInterrupts() {
    /* Disable further interrupts from arriving. */
  }
}
```

FIGURE 29. Example of an interaction class (ISR) in the cruise controller application.

`engine_isr`, and so on. The fields are annotated with `@InteractionCode` since we do not model them in the generated *Circus* application process and *OhCircus* class. Here, the ISRs essentially fire aperiodic (long) events that are bound to the aperiodic handlers of the mission. An example of a class for an interrupt service routine is included in Fig. 29. The constructor of the class is parametrised in terms of the `AperiodicLongEvent` instance that must be fired when the interrupt occurs, and the handler code defined by the `handle()` method simply fires this event prior to determining the cause of the interrupt (the actual code for this is omitted).

We observe that the class `EngineISR` is annotated with `@InteractionClass` to determine its membership to the category **IC** of interaction classes; in the model generation process, it is thus ignored. Likewise, the four methods `createEvents()`, `createISRs()`, `registerISRs()` and `unregisterISRs()` defined inside the `ACCMission` class (Fig. 28) are annotated with `@InteractionCode`. This tells the translator that we do not model these methods either.

Importantly, the implementation of the `EngineISR` class justifies the assumptions we make about atomicity of interactions, modelled by synchronisations. We have a call to `disableInterrupts()` at the beginning of the interrupt handler to disable further interrupts from the engine to guarantee atomicity of the interaction (the actual code for this is omitted for brevity). We can moreover think of a few appropriate design patterns for the ISRs, but ultimately it is the obligation of the engineer to validate the assumptions. Although this is an important issue in its own right, a detailed discussion is beyond the scope and contribution of this article.

Apart from identifying interaction code, we also have to annotate the `ACCMission` class with the identifier of the mission (here `ACCMId`). Likewise, all handlers have to be annotated with the identifier of the respective handler. An example of an annotated handler class is presented in Fig. B.1 in Appendix B. We have an annotation `@HandlerId` to specify the unique identifier of the handler. In addition, the `@BoundEvent` annotation determines the external event that the handler is bound to. We recapture it below.

```
@HandlerId("EngineHId")
@BoundEvent(value="engine", type="boolean")
public class Engine extends
    AperiodicLongEventHandler {
  ...
}
```

External events that release the handler are synchronisations on the channel *engine* of type *boolean*. Apart from this, the handler also contains interaction code that is not modelled. This is the method `enableInterrupts()`, which reenables the interrupts after they have been disabled by the interrupt service routine. This design ensures that no external event can intervene between the occurrence of an interrupt and release of the corresponding handler. As noted before, it justifies our assumption of atomic interactions. We note that this mechanism is essential for the correctness of the program in general. In other words, it was not introduced for reasons of applying our technique.

A second annotation we require is on one of the parameters of the constructor, namely `engine_event` of type `AperiodicLongEvent`. This is because we do not model SCJ event classes. In the program, they are used merely as a means to an end: whereas interrupts represent the actual external

```
@DeviceAccess("set_voltage~!~voltage \\then \\Skip")
private synchronized void writeVoltage() {
  final long SET_VOLTAGE_REG = 0x1234;
  try {
    RawIntegralAccess io_port =
      RawMemory.createRawIntegralInstance(
      RawMemory.IO_PORT_MAPPED, SET_VOLTAGE_REG, 1);
    /* Write out voltage to the throttle as a single byte. */
    io_port.setByte(0, (byte) (voltage * 10));
  }
  catch (Exception e) { }
}
```

FIGURE 30. External device access in the `ThrottleController` handler.

event being raised, instances of `AperiodicEvent` and `AperiodicLongEvent` are used to release the aperiodic handler that is associated with the event.

A more interesting example is the periodic handler `ThrottleController`. Here, we have an explicit external device access, namely to write out the voltage to the throttle. This is done by a method `writeVoltage()` from within the class. Fig. 30 recaptures the definition of this method. Most importantly, we have an annotation `@DeviceAccess` that specifies the action model for the device access: here an output prefix on *set_voltage*. We observe that the use of a `try/catch` statement in the method body is not in contradiction with the language constraints in Section 6.1.2 because we permit such statements in interaction code. Runtime exceptions should, however, not be thrown beyond the entry point of interaction methods. Program error exceptions may be thrown, though, and are modelled by the divergent action **abort**.

In summary, we conclude that specifying the annotations for the ACC inherently does not require notable expertise in the formal modelling notation. It merely requires the developer to identify code for interaction and device access. The annotation framework, due to its generality, also paves the way for custom extensions to treat other features of SCJ that are currently not catered for by our modelling approach. For instance, the `@InteractionCode` annotation may be used to specify the behaviour of code that utilises console I/O. The general benefit of our approach is that it allows us to abstract from the details of a piece of SCJ code (encapsulated in a method) by providing a specification of its behaviour as an action. Exploring this further is an interesting topic for future work.

7.3. Implementation Issues

The translator takes as its input the annotated SCJ program, as for now we do not provide any support for automating the annotation process. In order to match the high-level translation rules (Appendix C), the tool has to determine the superclasses and implemented interfaces of active classes, namely those annotated with `@ActiveData` and thus belonging to either **SMMC** or **HC**. We recall that our annotations do not encode the precise type of an SCJ application class. Specifically, the type can be extracted from the annotated syntax tree that is obtained using the Compiler API of the JDK 7. We extend this analysis further by verifying the constraints defined in Section 6.1 and the consistency of the annotations introduced. Inconsistencies or violations detected by the tool are reported to the user and abort the model generation process. Such failures normally reveal the need for refactoring the code and reviewing the annotations that have been inserted. The main challenge is to determine whether methods require action models or can be given a model as a data operation. In general, action models are required for the following types of methods.

1. Methods that are called by the SCJ infrastructure. These are typically methods that override classes or interfaces of the SCJ component framework.
2. Methods that carry out device or hardware accesses and, therefore, are correspondingly annotated.
3. Methods that call, directly or indirectly, other methods that requires an action model.

With regards to (1), we can easily determine methods that fall into this category by examining their name and signature. For (2), we can determine the relevant methods by probing type information in the annotated syntax tree of the program, as this information retains annotations defined on the various kinds of program elements. In order to determine methods of the third type above, we have to perform a control flow analysis based on the call-dependency of methods. For this, we have implemented a general utility to encode binary relations over arbitrary types and to facilitate efficient calculation of their closure. Hence, it is possible to automate the decision whether a method requires an action model, eliminating the need for annotations.

Model generation combines Stage (2) and Stage (3) of the translation process. For technical reasons, we do not rewrite the code in Stage (2), but directly define translation rules that take the rewrite rules in Fig. 6 into account during low-level statement translation. APIs

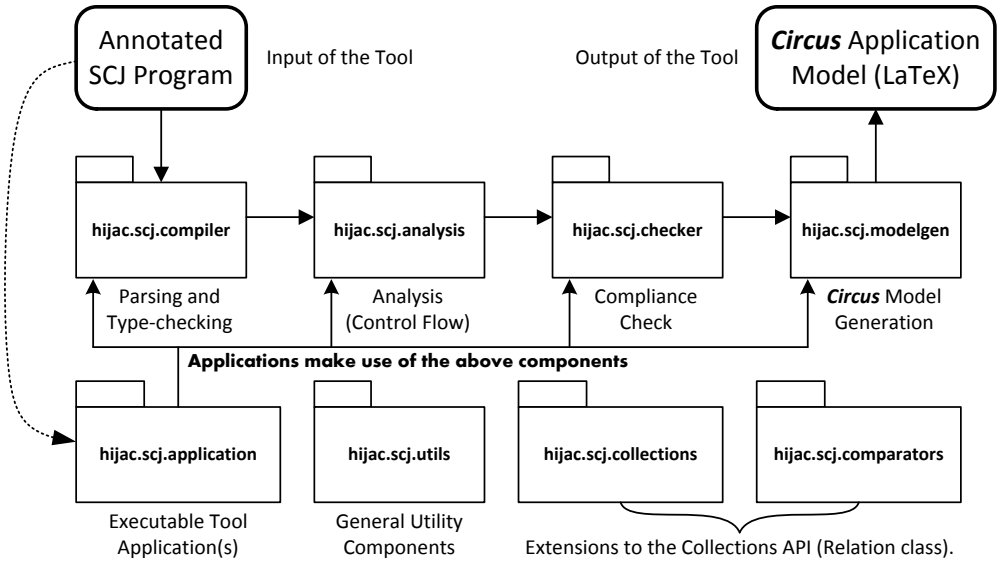

FIGURE 31. Top-level package view of your tool architecture.

for automatic code rewriting exist [49], but their use introduces unnecessary complexity in our tool.

Our tool consists of a number of components that perform essential processing tasks. They are illustrated in Fig. 31. The `compiler` package addresses the parsing and type-checking of the SCJ program; it uses existing JDK tools. The `analysis` package provides a facility for analysing the parsed program; it infers and records information that is later needed for the translation. The `checker` package performs a validity check of the SCJ program to ensure it belongs to the subset of translatable programs as defined in Section 6.1. The `modelgen` package provides components that carry out the actual generation of the model. Finally, apart from the four packages above, we also have the `application` package, which includes the top-level tool application, as well as the `utils` package, which provides various collections of utility functions. The remaining packages `collections` and `comparators` extend the Java collection API, most notably with an efficient implementation of mathematical relations.

The translator itself is implemented by virtue of a framework that realises a plugin architecture. The translator as a whole therefore consists of a collection of smaller translator plugins. Each translator plugin caters for the construction of a particular part of the model and exhibits a dependency on other translator plugins that have to be executed before it. Hence, we have, for instance, plugins that cater for the definition of channels, the introduction of axiomatic constants for handler and mission identifiers, and the construction of processes for active components and *OhCircus* classes for data objects. Translator plugins are provided by implementing an interface `Translator` in our tool framework: the implementation determines applicability of the translator, dependency to other translators, and the actual output produced, as well as the file(s) to which the output is appended.

High-level Translation. High-level translation implements the rules in Appendix C (there is a plugin for each rule). The plugins make use of string templates and the FreeMarker library. String templates, in general, offer a clean way of isolating static and dynamic aspects of any kind of textual patterns. Here, in particular, they offer traceability to the formal specification of the rules. Template models, which are certain kinds of classes of the FreeMarker API, make dynamic information available from within the templates that is needed by the rules, such as the functions IdOf, TypeOf or BoundEvents used in the right-hand side of the high-level translation rules. We take advantage here of the expressibility and extendibility of the FreeMarker template language to deal with special cases, and hence reduce the number of rules potentially needed.

Low-level Translation. Low-level translation realises the translation of statements, expressions, values and types. Although we specify these as templates as before, the translation is driven by visitors, which are used by the high-level translator plugins. Visitors are a general design pattern that supports the traversal of tree structures, and the JDK Compiler API comes with its own implementation of the pattern. We make use of two visitors, one that produces an action model and one that produces a data operation (*OhCircus* method).

Overall the translation process and tool seem to be robust and stable. Clean and sound software design principles, which isolate concerns like parsing, analysis, validation and translation into loosely-coupled components, ensure that the tool can be easily extended and modified, satisfying future needs of further elaborating our models by including features of SCJ that are currently not supported.

8. CONCLUSION

As far as we know, what we have presented here is the first formalisation of the SCJ paradigm. Our models capture the essence of its design, and are an essential asset for analysis and development techniques for SCJ programs based on refinement. There are a number of refinement-based techniques that are enabled by the availability of a formal model of SCJ in a state-rich process algebra. Particular techniques for *Circus* are reported in [21, 39, 50, 51], and techniques for languages of the same line are also reported in [52, 53, 54, 55]. The translation of the *Circus* model to work with tools supporting these techniques is a much easier exercise than the translation from the SCJ program directly.

Our models are in line with the SCJ technology specification in the absence of deadline-miss situations, which we do not consider. We hence assume that scheduling analysis and environmental assumptions about external interactions ensure that these situations essentially cannot occur. Imperfection or imprecision of timing evidence therefore already has to be accounted for at the level of scheduling analysis, which is currently a caveat for using our models.

A notable achievement is our solution to encode active class behaviour and data objects as independent dimensions. This increases the modularity of models and provides opportunities for modular reasoning, but also emphasises the need for an integrated formalism including constructs of *OhCircus* and *Circus* Time.

To validate the framework and application models, we have translated the respective *Circus* processes into pure CSP models, which we then submitted to the FDR [33] model checker. The CSP translation encapsulates all *Circus* state into process parameters. Timing aspects are ignored, and so is the detailed application-level behaviour of handlers. Apart from this, the CSP model retains exactly the structure of the corresponding *Circus* model. CSP processes, like *handlers_field* below, are used to encapsulate individual state components of a *Circus* process.

$$\begin{array}{l}\textbf{channel}\ handlers\_get : Set(HandlerId)\\ \textbf{channel}\ handlers\_set : Set(HandlerId)\\ handlers\_field(v) =\\ \quad handlers\_get\,!\,v \longrightarrow handlers\_field(v)\ \Box\\ \quad handlers\_set\,?\,v \longrightarrow handlers\_field(v)\end{array}$$

The CSP process $handlers\_field(v)$ encapsulates, for instance, the state component *handlers* of the process *MissionFW* in Fig. 10, where v determines its initial value. Channel communications on *handlers_get* and *handlers_set* are now used to read and write to the state component. For each state component of a *Circus* process, we define a CSP process similar in shape to $handlers\_field(v)$. Their parallel composition yields a model for the entire state, and that model is further composed with a translation of the local actions and main action of the *Circus* process. (Local definitions can be easily supported in CSP via **let** constructs.)

The resulting CSP process, after hiding the channels used for reading and modifying state components, exhibits precisely the behaviour of the original *Circus* process under CSP's failure-divergence semantics, and besides is amenable to model checking and, in addition, animation via tools such as FDR [33] and ProB [56].

Properties that we examined and validated are deadlock freedom and termination of several simplified application scenarios. Carrying out these checks for the entire ACC model would have been theoretically possible, but is thwarted by the complexity of the CSP models due to state explosion. Assertions of the following form have been used to establish termination.

$$\textbf{skip} \sqsubseteq_{FD} (System \llbracket Events \rrbracket TestEnv) \setminus Events$$

The *System* process refers to the process of the SCJ application, and *Events* are the channels used for external interactions. The *TestEnv* process provides a testing environment for the application scenario that interacts through the external channels.

Our validation efforts proved to be valuable in identifying subtle issues in earlier versions of our models. For instance, in the *MissionFW* process in Fig. 10, we formerly made use of an interrupt $(\ldots) \triangle initializeRet\,.\,mission \longrightarrow \textbf{skip}$ in the *Initialize* action to terminate that action. Model checking revealed a race condition that can result in a registered handler not being recorded in the state. Problems like these are difficult to diagnose *a posteriori* using testing-based validation approaches due to the amount of nondeterminism in SCJ program executions, as it naturally arises from the parallelism of handlers. Our models are essentially modular, and this enables and facilitates model checking to focus on particular parts of the model in isolation, both in terms of the framework and application processes.

More generally, there are nonetheless limitations to translating *Circus* into CSP. These apply, in particular, when modelling composite values and data operations. In our case, however, we could, model the entire SCJ framework. The feedback we obtained using FDR increases the confidence in our our models; this is further enhanced by the many discussions we had with experts in the SCJ technology.

To validate the tool, we have used it to generate the model of the ACC. This produces a *Circus* model that can be parsed and type-checked. The implementation of the tool *per se* supports our claim that models can be generated automatically. We are currently evaluating and testing the tool with further examples, in particular the collision detector in [36].

The direct correspondence between SCJ programs and our models enables automation in both directions: the framework processes are the same for all programs, and the application processes use a fixed modelling pattern. As part of our wider research agenda, we

are also developing a complementary tool that translates *Circus* models into SCJ programs. That tool will be useful if no implementation is *a priori* given, but can be designed *ad lib* from a specification. The reason we did not develop that tool first is that our current work is illuminating in identifying the features of SCJ that can be supported by our formalisms.

Related Work. Although there are many approaches and tools to reason about object-oriented programs and Java [57, 58], they do not cater for the specificities of concurrency in SCJ. Brooke et al. present a CSP specification for a concurrency model for Eiffel (SCOOP) [59]. Their CSP specification shares some basic ideas with our *Circus* models, but is necessarily more complex due to its generality. A recent work [60] examines test generation with strong coverage criteria; part of it is a formal specification of classes and methods in the Real-Time Java API.

Kalibera et al. [61] are concerned with scheduling analysis and race conditions in SCJ programs, but do not use proof-based techniques. Instead, exhaustive testing and model checking is applied.

Annotation-based techniques for SCJ can be found in [18, 62]. In [18] annotations are used to check for compliance with a particular level of SCJ, and for safe use of memory. Haddad et al. define SafeJML [62], which extends JML [57] to cover timing properties. It reuses existing technology for worst-case execution-time analysis in the context of SCJ. Our model is a conceivable candidate to justify the soundness of checks supported by the annotations and tools above.

Future Work. Our primary future work is to elaborate the model and translation to account for additional features of SCJ that for now have been ignored. This is, in particular, memory management, timing-related classes, and the support for active objects, that is, data objects that interact with either the hardware or framework. So far we have excluded this possibility as it did not arise in our examples, but one might envisage cases where support for such objects is desirable, like, for instance, a file class that represents data and at the same time interacts with a physical disk. We are currently considering a solution that provides a limited facility to model data objects as a combination of class and process, just like active components of the SCJ mission model. Additional restrictions to be imposed on such objects are still largely an open issue.

Another future work is to tackle issues of robustness by dealing with missed deadlines. SCJ offers support to detect and react to such situations in the program. Or models, however, currently do not formally capture this feature, mostly as it appears to be more challenging to integrate and we do not have SCJ applications that exploit it. Modelling it, however, enables the support for a larger class of implementations.

Our structural constraints restrict program designs. In particular, they prevent us from making effective use of subclassing where SCJ components are involved. This may have implications on modularisation and reuse of verification arguments, for instance, in view of product lines. We also hope to address this issue in future research once more experience has been gained in constructing verification arguments.

Our long term goal is the definition of refinement-based techniques for SCJ program development. Like in the *Circus* standard technique [21], we will devise a refinement strategy to transform centralised abstract *Circus* Time models into an SCJ model as described here. The development of this strategy, and the proof of the refinement rules that it will require are a challenging aspect of this endeavour; initial results have been published in [35]. This involves the identification of refinement and modelling patterns. All this shall also provide further practical validation of our model.

ACKNOWLEDGEMENTS

We are grateful to Chris Marriott, Kun Wei, and Jim Woodcock for useful discussions of our models. This work is funded by the EPSRC grant EP/H017461/1.

REFERENCES

[1] Søndergaard, H., Thomsen, B., and Ravn, A. P. (2006) A Ravenscar-Java Profile Implementation. *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2006*, Paris, France, 11–13 October, pp. 38–47. ACM, New York, NY, USA.

[2] The Open Group (2011) Safety Critical Java Technology Specification. Technical Report JSR-302. Java Community Process. Available from `http://jcp.org/aboutJava/communityprocess/edr/jsr302/`.

[3] Cinnober Stockholm, Sweden (2012). The benefits of using Java as a high-performance language for mission critical financial applications. White paper available at `http://www.cinnober.com/news/benefits-using-java-highperformance-language`.

[4] atego (2012). Aonix Perc Pico. Product information and datasheet available at `http://www.atego.com/products/aonix-perc-pico/`.

[5] DO-332 (2011) Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A. Technical Report. RTCA Inc., Washington, DC, USA. Available at `http://www.rtca.org`.

[6] DO-178C (2011) Software Considerations in Airborne Systems and Equipment Certification. Technical Report. RTCA Inc., Washington, DC, USA. Available at `http://www.rtca.org`.

[7] Walter, A. (2010) Towards Certification of Java Applications for Safety Critical Projects. *Proceedings of Embedded Real-Time Software and Systems, ERTS^2 2010*, Toulouse, France, 19–21 May, pp. 1–7. Available online at `http://web1.see.asso.fr/erts2010/Default.aspx-Id=973-Idd=982.htm`.

[8] Hunt, J. (2010) Realtime Java Technology in Avionics Systems. *Proceedings of the 8th International*

Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2010, Prague, Czech Republic, 19–21 August, pp. 138–147. ACM, New York, NY, USA.

[9] NIST Special Publication 500-243 (2000) Requirements for Real-time Extensions for the Java Platform: Report from the Requirement Group for Real-time Extension for the Java Platform. Technical Report. National Institute for Standards and Technology, Gaithersburg, MD 20899-1070, USA. Available at `http://www.itl.nist.gov/lab/specpubs/sp500.htm`.

[10] Wellings, A. (2004) *Concurrent and Real-Time Programming in Java*. Wiley, Hoboken, NJ, USA.

[11] Tofte, M. and Talpin, J.-P. (1997) Region-Based Memory Management. *Information and Computation*, **132**, 109–176.

[12] Armbruster, A., Baker, J., Cunei, A., Flack, C., Holmes, D., Pizlo, F., Pla, E., Prochazka, M., and Vitek, J. (2007) A Real-Time Java Virtual Machine with Applications in Avionics. *ACM Transactions on Embedded Computing Systems*, **7**, 5:1–5:49.

[13] Pizlo, F., Ziarek, L., and Vitek, J. (2009) Real Time Java on resource-constrained platforms with Fiji VM. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009*, Madrid, Spain, 23–25 September, pp. 110–119. ACM, New York, NY, USA.

[14] aicas GmbH Karlsruhe, Germany (2012) *JamaicaVM 6.2 — User Manual. Java Technology for Critical Embedded Systems*. Available at `http://www.aicas.com/jamaica.html`.

[15] Bruno, E. J. and Bollella, G. (2009) *Real-Time Java^{TM} Programming with Java RTS*. Prentice Hall, Upper Saddle River, NJ, USA.

[16] IBM (2012). WebSphere Real Time. Product information available at `http://www-03.ibm.com/software/products/us/en/real-time`.

[17] Henties, T., Hunt, J., Locke, D., Nilsen, K., Schoeberl, M., and Vitek, J. (2009) Java for Safety-Critical Applications. *Proceedings of the 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, UK, 29 March.

[18] Tang, D., Plsek, A., and Vitek, J. (2010) Static Checking of Safety Critical Java Annotations. *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2010*, Prague, Czech Republic, 19–21 August, pp. 148–154. ACM, New York, NY, USA.

[19] Burns, A. (1999) The Ravenscar Profile. *ACM SIGAda Ada Letters*, **XIX**, 49–52.

[20] Woodcock, J. and Cavalcanti, A. (2001) A Concurrent Language for Refinement. *Proceedings of the 5th Irish Workshop on Formal Methods, IW-FM'01*, Dublin, Ireland, 16–17 July, pp. 93–115. BCS, Swindon, UK.

[21] Cavalcanti, A., Sampaio, A., and Woodcock, J. (2003) A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, **15**, 146–181.

[22] Spivey, J. M. (1992) *The Z Notation: A Reference Manual*. Prentice Hall, Upper Saddle River, NJ, USA.

[23] Hoare, C. A. R. (1985) *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, NJ, USA.

[24] Morgan, C. C. (1998) *Programming from Specifications*. Prentice Hall, Upper Saddle River, NJ, USA.

[25] Oliveira, M., Cavalcanti, A., and Woodcock, J. (2009) A UTP semantics for *Circus*. *Formal Aspects of Computing*, **21**, 3–32.

[26] Cavalcanti, A., Clayton, P., and O'Halloran, C. (2011) From control law diagrams to Ada via *Circus*. *Formal Aspects of Computing*, **23**, 465–512.

[27] Marriott, C., Zeyda, F., and Cavalcanti, A. (2012) A Tool Chain for the Automatic Generation of *Circus* Specifications of Simulink Diagrams. *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z, ABZ 2012*, Pisa, Italy, 18–21 June, LNCS, **7316**, pp. 294–307. Springer-Verlag, Berlin.

[28] Hoare, C. A. R. and Jifeng, H. (1998) *Unifying Theories of Programming*. Prentice Hall, Upper Saddle River, NJ, USA.

[29] Cavalcanti, A., Sampaio, A., and Woodcock, J. (2005) Unifying classes and processes. *Software and Systems Modeling*, **4**, 277–296.

[30] Cavalcanti, A., Wellings, A., and Woodcock, J. (2011) The Safety-Critical Java Memory Model: A Formal Account. *Proceedings of the 17th International Symposium of Formal Methods, FM 2011*, Limerick, Ireland, 20–24 June, LNCS, **6664**, pp. 246–261. Springer-Verlag, Berlin.

[31] Wellings, A. and Kim, M. (2010) Asynchronous Event Handling and Safety Critical Java. *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2010*, Prague, Czech Republic, 19–21 August, pp. 53–62. ACM, New York, NY, USA.

[32] Issue 4.8 (2008) SPARK 95 – The SPADE Ada 95 Kernel. Technical Report. Praxis High Integrity Systems Ltd, Bath, BA1 1PX, UK.

[33] Formal Systems (Europe) Ltd Oxford, UK (2010) *Failures-Divergence Refinement, FDR2 User Manuel*. Available from `http://www.fsel.com/documentation/fdr2/fdr2manual.pdf`.

[34] Malik, P. and Utting, M. (2005) CZT: A Framework for Z Tools. *Proceedings of the 4th International Conference of Z and B Users, ZB 2005*, Guildford, UK, 13–15 April, LNCS, **3455**, pp. 65–84. Springer-Verlag, Berlin. See also `http://czt.sourceforge.net`.

[35] Zeyda, F., Cavalcanti, A., Wellings, A., Woodcock, J., and Wei, K. (2012) Refinement of the Parallel CD_x. Technical report. University of York, York, UK. Available from `http://www.cs.york.ac.uk/circus/publications/techreports/index.html`.

[36] Cavalcanti, A., Zeyda, F., Wellings, A., Woodcock, J., and Wei, K. (2013) Safety-critical Java programs from *Circus* models. *Real-Time Systems*, **Currently under publication, no volume and page number yet.**

[37] Zeyda, F., Cavalcanti, A., and Wellings, A. (2011) The Safety-Critical Java Mission Model: A Formal Account. *Proceedings of the 13th International Conference on Formal Engineering Methods, ICFEM 2011*, Durham, UK, 26–28 October, LNCS, **6991**, pp. 49–65. Springer-Verlag, Berlin.

[38] Hatley, D. J. and Pirbhai, I. A. (1987) *Strategies for Real-Time System Specification*. Dorset House Publishing, New York, NY 10014, USA.

[39] Oliveira, M. (2005) Formal Derivation of State-Rich Reactive Programs using *Circus*. PhD thesis Department of Computer Science, University of York York, UK.

[40] Dijkstra, E. W. (1976) *A Discipline of Programming*. Prentice Hall, Upper Saddle River, NJ, USA.

[41] Sherif, A., Cavalcanti, A., Jifeng, H., and Sampaio, A. (2010) A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, **22**, 153–191.

[42] Bouyer, P., Larsen, K. G., Markey, N., Sankur, O., and Thrane, C. (2011) Timed Automata Can Always Be Made Implementable. *Proceedings of the 22nd International Conference on Concurrency Theory, CONCUR 2011*, Aachen, Germany, 5–10 September, LNCS, **6901**, pp. 76–91. Springer-Verlag, Berlin.

[43] Cavalcanti, A., Wellings, A., Woodcock, J., Wei, K., and Zeyda, F. (2011) Safety-Critical Java in *Circus*. *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2011*, York, UK, 26–28 September, pp. 20–29. ACM, New York, NY, USA.

[44] Gosling, J., Joy, B., Steele, G. L., and Bracha, G. (2005) *Java (TM) Language Specification, Third Edition*. Addison-Wesley Professional, Boston, MA, USA.

[45] Oliveira, W. R. and Barros, R. S. M. (1997) The Real Numbers in Z. *Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop*, Ilkley, UK, 14–15 July, pp. 1–15. BCS, Swindon, UK.

[46] Oracle California, USA (2011) *Java Platform Standard Edition 7 Documentation*. Available at `http://docs.oracle.com/javase/7/docs/`.

[47] Ernst, M. D. (2013) Type Annotations Specification. Technical Report JSR-308. Java Community Process. Available at `http://jcp.org/aboutJava/communityprocess/edr/jsr308/`.

[48] FreeMarker Project (2012) *FreeMarker Manual*, 2.3.19 edition. Available at `http://freemarker.org/docs/`.

[49] Fuhrer, R. M., Keller, M., and Kiezun, A. (2007) Advanced Refactoring in the Eclipse JDT: Past, Present, and Future. *Proceedings of the First Workshop on Refactoring Tools, WRT 2007*, Berlin, Germany, 30 July–3 August, pp. 30–31.

[50] Zeyda, F. and Cavalcanti, A. (2010) Automating Refinement of *Circus* Programs. *Proceedings of the 13th Brazilian Symposium on Formal Methods, SBMF 2010*, Natal, Brazil, 8–11 November, LNCS, **6527**, pp. 274–290. Springer-Verlag, Berlin.

[51] Zeyda, F., Oliveira, M., and Cavalcanti, A. (2012) Mechanised support for sounds refinement tactics. *Formal Aspects of Computing*, **24**, 127–160.

[52] Abrial, J.-R. (2007) A System Development Process with Event-B and the Rodin Platform. *Proceedings of the 9th International Conference on Formal Engineering Methods, ICFEM 2007*, Florida, USA, 14–15 November, LNCS, **4789**, pp. 1–3. Springer-Verlag, Berlin.

[53] Fischer, C. (1997) CSP-OZ: A Combination of Object-Z and CSP. *Proceedings of FMOODS'97, IFIP TC6 WG6.1 International Conference on Formal Methods for Open Object-based Distributed Systems*, Canterbury, UK, 21–23 July, pp. 423–438. Chapman & Hall Ltd., London, UK.

[54] Treharne, H. and Schneider, S. (2000) How to Drive a B Machine. *Proceedings of the First International Conference of B and Z Users, ZB 2000*, York, UK, 29 August–2 September, LNCS, **1878**, pp. 188–208. Springer-Verlag, Berlin.

[55] Schneider, S., Treharne, H., and Wehrheim, H. (2010) A CSP Approach to Control in Event-B. *Proceedings of the 8th International Conference on Integrated Formal Methods, IFM 2010*, Nancy, France, 11–14 October, LNCS, **6396**, pp. 260–274. Springer-Verlag, Berlin.

[56] Leuschel, M. and Butler, M. (2008) ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, **10**, 185–203.

[57] Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005) An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, **7**, 212–232.

[58] Beckert, B., Hähnle, R., and Schmitt, P. H. (2007) *Verification of Object-Oriented Software. The KeY Approach*, LNCS, **4334**. Springer-Verlag, Berlin.

[59] Brooke, P. J., Paige, R., and Jacob, J. (2007) A CSP model of Eiffel's SCOOP. *Formal Aspects of Computing*, **19**, 487–512.

[60] Ahrendt, W., Mostowski, W., and Paganelli, G. (2012) Real-time Java API Specifications for High Coverage Test Generation. *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2012*, Copenhagen, Denmark, 24–26 October, pp. 145–154. ACM, New York, NY, USA.

[61] Kalibera, T., Parizek, P., Malohlava, M., and Schoeberl, M. (2010) Exhaustive Testing of Safety Critical Java. *Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES 2010*, Prague, Czech Republic, 19–21 August, pp. 164–174. ACM, New York, NY, USA.

[62] Haddad, G., Hussain, F., and Leavens, G. T. (2010) The Design of SafeJML, A Specification Language for SCJ with Support for WCET Specification. *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2010*, Prague, Czech Republic, 19–21 August, pp. 155–163. ACM, New York, NY, USA.

APPENDIX A. CHANNELS USED IN THE CRUISE CONTROLLER MODEL

Channel	Type	Description
wheel_shaft	–	occurs with each rotation of the wheel shaft
engine_on	–	engine is switched on
engine_off	–	engine is switched off
brake	*boolean*	brake pedal is pressed or released
top_gear_engaged	–	driver switches into top gear
top_gear_disengaged	–	driver switches out of top gear
lever	*LEVER*	driver operates the command lever
set_voltage	0..80	sets the voltage on the throttle actuator

TABLE A.1. Channels for external events of the cruise controller model.

Method Channel(s)	Class or Interface	Target	Param	Return
setUp[*Call*/*Ret*]	`Safelet`	–	–	–
tearDown[*Call*/*Ret*]	`Safelet`	–	–	–
getNextMission[*Call*/*Ret*]	`MissionSequencer`	–	–	*MissionId*
initialize[*Call*/*Ret*]	`Mission`	*MissionId*	–	–
cleanup[*Call*/*Ret*]	`Mission`	*MissionId*	–	–
requestTermination[*Call*/*Ret*]	`Mission`	–	–	–
terminationPending[*Call*/*Ret*]	`Mission`	–	–	*boolean*
register	`Managed[Long]EventHandler`	*HandlerId*	–	–

Above *Chan*[*Call*/*Ret*] abbreviates a channel pair *ChanCall* and *ChanRet*.

TABLE A.2. Channel pairs for infrastructure methods.

The types of the channels in Table A.2 are determined by the classes that include the methods, and the types of their possible parameters and return values. The purpose of the target parameter is to disambiguate the use of the channel in the context of multiple objects. If there are no parameters or return values, the types simplify to *Target*. And if the method is only called with the same object as a target, the channels become typeless thus representing simple synchronisations.

APPENDIX B. SCJ PROGRAM CODE

```
@HandlerId("EngineHId")
@BoundEvent(channel = "engine", type = "boolean")
public class Engine extends AperiodicLongEventHandler {
  private Controller cruise;

  public Engine(Controller cruise, @Ignore AperiodicLongEvent engine_event) {
    super(...);
    this.cruise = cruise;
  }

  public void handleAsyncLongEvent(long param) {
    int event = (int) param;
    switch (event) {
      case ENGINE_ON:
        cruise.engineOn();
        break;

      case ENGINE_OFF:
        cruise.engineOff();
        break;
    }
    enableInterrupts();
  }

  @InteractionCode
  public void enableInterrupts() {
    /* Program code to re-enable interrupts. */
  }
}
```

FIGURE B.1. Annotated version of the `Engine` handler class.

APPENDIX C. HIGH-LEVEL TRANSLATION RULES

We use three auxiliary functions in the high-level translation rules for SCJ components.

1. $\mathsf{IdOf}(type)$ yields the identifier of a mission or handler class;
2. $\mathsf{TypeOf}(obj)$ infers the type of a Java object; and,
3. $\mathsf{BoundEvents}(type)$ determines the external events bound to an aperiodic event handler.

$$
\begin{array}{l}
[\![\texttt{class } \underline{CName} \texttt{ implements Safelet \{} \\
\quad \underline{FieldDecls} \\
\quad \texttt{public } \underline{CName}\texttt{() \{ } \underline{InitStmts} \texttt{ \}} \\
\quad \underline{AMethDecl1} \\
\quad \underline{AMethDecl2} \\
\quad \ldots \\
\quad \texttt{public MissionSequencer getSequencer() \{} \\
\quad\quad \texttt{return new } \underline{MSName}\texttt{();} \\
\quad \texttt{\}} \\
\texttt{\}} \;]\!]
\end{array}
\;\widehat{=}\;
\begin{array}{l}
\textbf{process } [\![\underline{CName}]\!]_{Name}\, App \;\widehat{=}\; \textbf{begin} \\
\quad \textbf{state } State \;==\; [this : [\![\underline{CName}]\!]_{Name}\, Class] \\
\quad Init \;==\; [State' \mid this' = \textbf{new } [\![\underline{CName}]\!]_{Name}\, Class] \\
\quad [\![\underline{AMethDecl1}]\!]_{AMethod} \\
\quad [\![\underline{AMethDecl2}]\!]_{AMethod} \\
\quad \ldots \\
\quad Methods \;\widehat{=}\; \\
\qquad \left(\mu X \bullet \left(\begin{array}{l} [\![\underline{AMethDecl1}]\!]_{Name}\, Meth \;\Box \\ [\![\underline{AMethDecl2}]\!]_{Name}\, Meth \;\Box \\ \ldots \end{array} \right) ;\; X \right) \\
\quad \bullet\; Init \;;\; (Methods \;\triangle\; end\_safelet\_app \longrightarrow \textbf{skip}) \\
\textbf{end}
\end{array}
$$

FIGURE C.1. Translation rule for classes implementing the `Safelet` interface.

$$
\left[\!\!\left[
\begin{array}{l}
\texttt{class } \underline{CName} \texttt{ extends MissionSequencer \{} \\
\quad \underline{FieldDecls} \\
\quad \texttt{public } \underline{CName}\texttt{() \{ } \underline{InitStmts} \texttt{ \}} \\
\quad \underline{AMethDecl1} \\
\quad \underline{AMethDecl2} \\
\quad \ldots \\
\texttt{\}}
\end{array}
\right]\!\!\right] \mathrel{\widehat{=}}
$$

$$
\begin{array}{l}
\textbf{process } [\![\underline{CName}]\!]_{Name}\, App \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state } State == [this : [\![\underline{CName}]\!]_{Name}\, Class] \\
\quad Init == [State' \mid this' = \textbf{new } [\![\underline{CName}]\!]_{Name}\, Class] \\
\quad [\![\underline{AMethDecl1}]\!]_{AMethod} \\
\quad [\![\underline{AMethDecl2}]\!]_{AMethod} \\
\quad \ldots \\
\quad Methods \mathrel{\widehat{=}} \\
\qquad \left(\mu X \bullet \left(\begin{array}{l} [\![\underline{AMethDecl1}]\!]_{Name}\, Meth \ \Box \\ [\![\underline{AMethDecl2}]\!]_{Name}\, Meth \ \Box \\ \ldots \end{array} \right) ; \ X \right) \\
\quad \bullet Init ; \ (Methods \triangle end\_sequencer\_app \longrightarrow \textbf{skip}) \\
\textbf{end}
\end{array}
$$

FIGURE C.2. Translation rule for classes extending the `MissionSequencer` class.

$$
\left[\!\!\left[
\begin{array}{l}
\texttt{class } \underline{CName} \texttt{ extends Mission \{} \\
\quad \underline{FieldDecls} \\
\quad \texttt{public } \underline{CName}\texttt{() \{ } \underline{InitStmts} \texttt{ \}} \\
\quad \underline{AMethDecl1} \\
\quad \underline{AMethDecl2} \\
\quad \ldots \\
\quad \texttt{public int missionMemorySize() \{} \\
\qquad \texttt{return } \underline{Expr} \texttt{ ;} \\
\quad \texttt{\}} \\
\texttt{\}}
\end{array}
\right]\!\!\right] \mathrel{\widehat{=}}
$$

$$
\begin{array}{l}
\textbf{process } [\![\underline{CName}]\!]_{Name}\, App \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state } State == [this : [\![\underline{CName}]\!]_{Name}\, Class] \\
\quad Init \mathrel{\widehat{=}} [\![\underline{CName}]\!]_{Name}\, Init\,?\, obj \longrightarrow this := obj \\
\quad [\![\underline{AMethDecl1}]\!]_{AMethod} \\
\quad [\![\underline{AMethDecl2}]\!]_{AMethod} \\
\quad \ldots \\
\quad Methods \mathrel{\widehat{=}} \\
\qquad \left(\mu X \bullet \left(\begin{array}{l} [\![\underline{AMethDecl1}]\!]_{Name}\, Meth \ \Box \\ [\![\underline{AMethDecl2}]\!]_{Name}\, Meth \ \Box \\ \ldots \end{array} \right) ; \ X \right) \\
\quad \bullet \left(\begin{array}{l} \mu X \bullet Init; \\ \left(Methods \triangle \ end\_mission\_app\,.\, IdOf(\underline{CName}) \longrightarrow X \right) \end{array} \right) \\
\qquad \triangle \ end\_mission\_fw \longrightarrow \textbf{skip} \\
\textbf{end}
\end{array}
$$

FIGURE C.3. Translation rule for classes extending the `Mission` class.

$$
\left[\!\!\left[
\begin{array}{l}
\texttt{class } \underline{CName} \texttt{ extends} \\
\quad \texttt{AperiodicEventHandler \{} \\
\quad \underline{FieldDecls} \\
\quad \texttt{public } \underline{CName}\texttt{() \{ } \underline{InitStmts} \texttt{ \}} \\
\quad \underline{AMethDecl1} \\
\quad \underline{AMethDecl2} \\
\quad \ldots \\
\quad \texttt{public void handleAsyncEvent() \{} \\
\qquad \underline{HdlStmts} \\
\quad \texttt{\}} \\
\texttt{\}}
\end{array}
\right]\!\!\right] \mathrel{\widehat{=}}
$$

$$
\begin{array}{l}
\textbf{process } [\![\underline{CName}]\!]_{Name}\, App \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state } State == [this : [\![\underline{CName}]\!]_{Name}\, Class] \\
\quad Init \mathrel{\widehat{=}} [\![\underline{CName}]\!]_{Init}\, ?\, obj \longrightarrow this := obj \\
\quad [\![\underline{AMethDecl1}]\!]_{AMethod} \\
\quad [\![\underline{AMethDecl2}]\!]_{AMethod} \\
\quad \ldots \\
\quad Methods \mathrel{\widehat{=}} \\
\qquad \left(\begin{array}{l} [\![\underline{AMethDecl1}]\!]_{Name}\, Meth \ \Box \\ [\![\underline{AMethDecl2}]\!]_{Name}\, Meth \ \Box \\ \ldots \end{array} \right) \\
\quad handleAsyncEvent \mathrel{\widehat{=}} \textbf{val } inp \bullet [\![\underline{HdlStmts}]\!]_{HdlBody} \\
\quad Execute \mathrel{\widehat{=}} enter\_dispatch\,.\, IdOf(\underline{CName}) \longrightarrow Dispatch \\
\quad Dispatch \mathrel{\widehat{=}} \\
\qquad \left(\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} Methods \ \Box \\ \left(\begin{array}{l} \Box \ evt : \mathsf{BoundEvents}(\underline{CName}) \bullet \\ evt\,?\, inp \longrightarrow handleAsyncEvent(inp) \end{array} \right) \end{array} \right) ; \ X \\ \Box \\ leave\_dispatch\,.\, IdOf(\underline{CName}) \longrightarrow \textbf{skip} \end{array} \right) \right) \\
\quad \bullet (\mu X \bullet Init ; \ Execute ; \ X) \triangle end\_mission\_fw \longrightarrow \textbf{skip} \\
\textbf{end}
\end{array}
$$

FIGURE C.4. Translation rule for classes extending the `AperiodicEventHandler` class.

$$
\left[\!\!\left[
\begin{array}{l}
\texttt{class } \underline{CName} \texttt{ extends} \\
\quad \texttt{PeriodicEventHandler \{} \\
\quad \underline{FieldDecls} \\
\quad \texttt{public } \underline{CName}\texttt{() \{ } \underline{InitStmts} \texttt{ \}} \\
\quad \underline{AMethDecl1} \\
\quad \underline{AMethDecl2} \\
\quad \ldots \\
\quad \texttt{public void handleAsyncEvent() \{} \\
\quad\quad \underline{HdlStmts} \\
\quad \texttt{\}} \\
\texttt{\}}
\end{array}
\right]\!\!\right]
\mathrel{\widehat{=}}
$$

process $[\![\underline{CName}]\!]_{Name}\, App \mathrel{\widehat{=}}$ **begin**

state $State == [this : [\![\underline{CName}]\!]_{Name}\, Class]$

$Init \mathrel{\widehat{=}} [\![\underline{CName}]\!]_{Name}\, Init \,?\, obj \longrightarrow this := obj$

$[\![\underline{AMethDecl1}]\!]_{AMethod}$

$[\![\underline{AMethDecl2}]\!]_{AMethod}$

$\ldots$

$Methods \mathrel{\widehat{=}}$

$$\left(\begin{array}{l} [\![\underline{AMethDecl1}]\!]_{Name}\, Meth \;\Box \\ [\![\underline{AMethDecl2}]\!]_{Name}\, Meth \;\Box \\ \ldots \end{array} \right)$$

$handleAsyncEvent \mathrel{\widehat{=}}$ **val** $inp \bullet [\![\underline{HdlStmts}]\!]_{HdlBody}$

$Execute \mathrel{\widehat{=}} enter\_dispatch\,.\,IdOf(\underline{CName}) \longrightarrow$
$(Dispatch \,[\![\{this\} \mid \{\!| release\_handler |\!\} \mid \varnothing]\!]\, Release)$

$Dispatch \mathrel{\widehat{=}}$

$$\left(\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} Methods \;\Box \\ \left(\begin{array}{l} release\_handler\,.\,IdOf(\underline{CName}) \longrightarrow \\ handleAsyncEvent \end{array} \right) \end{array} \right) ;\; X \\ \Box \\ leave\_dispatch\,.\,IdOf(\underline{CName}) \longrightarrow \textbf{skip} \end{array} \right) \right)$$

$Release \mathrel{\widehat{=}}$

$$\left(\mu X \bullet \left(\begin{array}{l} release\_handler\,.\,IdOf(\underline{CName}) \longrightarrow \textbf{skip}; \\ \textbf{wait } this.period \end{array} \right) ;\; X \right) \triangle\; leave\_dispatch\,.\,IdOf(\underline{CName}) \longrightarrow \textbf{skip}$$

$\bullet\; (\mu X \bullet Init \;;\; Execute \;;\; X) \triangle end\_mission\_fw \longrightarrow \textbf{skip}$

end

FIGURE C.5. Translation rule for classes extending the `PeriodicEventHandler` class.

APPENDIX D. LOW-LEVEL TRANSLATION RULES

Rule	Application	Result	Addendum
TR1	$[\![\texttt{boolean}]\!]_{Type}$	*boolean*	where *boolean* ::= *TRUE* | *FALSE*
TR2	$[\![\texttt{byte}]\!]_{Type}$	*byte*	where $byte \mathrel{\widehat{=}} -128 \,..\, 127$
TR3	$[\![\texttt{short}]\!]_{Type}$	*short*	where $short \mathrel{\widehat{=}} -2^{15} \,..\, 2^{15}-1$
TR4	$[\![\texttt{int}]\!]_{Type}$	*int*	where $int \mathrel{\widehat{=}} -2^{31} \,..\, 2^{31}-1$
TR5	$[\![\texttt{long}]\!]_{Type}$	*long*	where $long \mathrel{\widehat{=}} -2^{63} \,..\, 2^{63}-1$
TR6	$[\![\texttt{char}]\!]_{Type}$	*char*	where $char \mathrel{\widehat{=}} 0 \,..\, 2^{16}-1$
TR7	$[\![\texttt{float}]\!]_{Type}$	*float*	where *float* is a given type
TR8	$[\![\texttt{double}]\!]_{Type}$	*double*	where *double* is a given type
TR9	$[\![\underline{RefType}]\!]_{Type}$	$[\![\underline{RefType}]\!]_{Name}\, Class$	where $\underline{RefType}$ is a reference type

TABLE D.1. Translation rules for Java types.

Rule	Application	Result	Note
VR1	$[\![\texttt{0 , 1 , 2 , ...}]\!]_{Value}$	0 , 1 , 2 , ...	unsigned numbers
VR2	$[\![\texttt{0 , -1 , -2 , ...}]\!]_{Value}$	0 , −1 , −2 , ...	signed numbers
VR3	$[\![\texttt{true}]\!]_{Value}$	*TRUE*	boolean value true
VR4	$[\![\texttt{false}]\!]_{Value}$	*FALSE*	boolean value false
VR5	$[\![\texttt{'}\underline{c}\texttt{'}]\!]_{Value}$	$\mathrm{ord}(\underline{c})$	where $\mathrm{ord}(\underline{c})$ gives the unicode of character $\underline{c}$
VR6	$[\![\texttt{"Foo"}]\!]_{Value}$	$\langle [\![\texttt{'F'}]\!]_{Value}, [\![\texttt{'o'}]\!]_{Value}, [\![\texttt{'o'}]\!]_{Value} \rangle$	strings are encoded as sequences
VR7	$[\![\texttt{null}]\!]_{Value}$	**null**	encodes null-reference for data objects

TABLE D.2. Translation rules for literal Java values.

Rule	Application	Result	Note
ER1	$[\![\underline{Value}]\!]_{Expr}$	$[\![\underline{Value}]\!]_{Value}$	where $\underline{Value}$ is a literal value
ER2	$[\![\underline{UnOp}\ \underline{Expr}]\!]_{Expr}$	$[\![\underline{UnOp}]\!]_{Expr}\ [\![\underline{Expr}]\!]_{Expr}$	where $\underline{UnOp}$ is a unary operator
ER3	$[\![\underline{Expr}_1\ \underline{BinOp}\ \underline{Expr}_2]\!]_{Expr}$	$[\![\underline{Expr}_1]\!]_{Expr}\ [\![\underline{BinOp}]\!]_{Expr}\ [\![\underline{Expr}_2]\!]_{Expr}$	where $\underline{BinOp}$ is a binary operator
ER4	$[\![\texttt{+}]\!]_{Expr}, [\![\texttt{-}]\!]_{Expr}, [\![\texttt{*}]\!]_{Expr}, \ldots$	$+, -, *, \ldots$	arithmetic operators
ER5	$[\![\texttt{\&\&}]\!]_{Expr}, [\![\texttt{||}]\!]_{Expr}, [\![\texttt{!}]\!]_{Expr}$	$\wedge, \vee, \neg$	translation of logical operators
ER6	$[\![\underline{Expr}\ \texttt{?}\ \underline{Expr}_1\ \texttt{:}\ \underline{Expr}_2]\!]_{Expr}$	**if** $[\![\underline{Expr}]\!]_{Expr} = TRUE$ **then** $[\![\underline{Expr}_1]\!]_{Expr}$ **else** $[\![\underline{Expr}_2]\!]_{Expr}$	translation of conditional expressions
ER7	$[\![\underline{Obj}.\underline{Meth}(\underline{Args})]\!]_{Expr}$	$[\![\underline{Obj}]\!]_{Expr} \cdot [\![\underline{Meth}]\!]_{Name}([\![\underline{Args}]\!]_{Expr})$	side-effect free call on a data object
ER8	$[\![(\underline{Arg1}, \underline{Arg2}, \ldots)]\!]_{Expr}$	$([\![\underline{Arg1}]\!]_{Expr}, [\![\underline{Arg2}]\!]_{Expr}, \ldots)$	translation of argument tuples

TABLE D.3. Translation rules for Java expressions.

Rule	Application	Result
FR1	$\left[\!\!\left[\begin{array}{l} \underline{FieldDecl} \\ \underline{FieldDecls} \end{array} \right]\!\!\right]_{FieldDecls}$ where $\underline{FieldDecls}$ is a non-empty list of field declarations	$[\![\underline{FieldDecl}]\!]_{Fields}$; $[\![\underline{FieldDecls}]\!]_{Fields}$
FR2	$[\![\underline{AccessModifier}\ \underline{FieldType}\ \underline{FieldName}\ \texttt{;}\ ]\!]_{Fields}$	$[\![\underline{FieldName}]\!]_{Name} : [\![\underline{FieldType}]\!]_{Type}$
FR3	$[\![\underline{AccessModifier}\ \underline{FieldType}\ \underline{FieldName}\ \texttt{=}\ \underline{Expr}\texttt{;}\ ]\!]_{Fields}$	$[\![\underline{FieldName}]\!]_{Name} : [\![\underline{FieldType}]\!]_{Type}$

We note that in **FR3**, we ignore the expression $\underline{Expr}$ initialising the variable since at this point we only consider the default initialisation of variables; the function $[\![\ldots]\!]_{FieldInit}$ deals with explicit initialisations separately.

TABLE D.4. Translation rules for field declarations.

Rule	Application	Result
SR7	$[\![\texttt{;}\ ]\!]_{Stmts}$ (empty list of statements)	**skip**
SR8	$[\![\underline{Stmts}_1\ \texttt{;}\ \underline{Stmts}_2]\!]_{Stmts}$	$[\![\underline{Stmts}_1]\!]_{Stmts}\ ;\ [\![\underline{Stmts}_2]\!]_{Stmts}$
SR9	$[\![\texttt{\{}\ \underline{Stmts}\ \texttt{\}}]\!]_{Stmts}$ (statement block)	$[\![\underline{Stmts}]\!]_{Stmts}$
SR10	$[\![\texttt{for (}\underline{Stmts}_1\texttt{;}\ \underline{Expr}\texttt{;}\ \underline{Stmts}_2\texttt{)}\ \underline{Stmts}]\!]_{Stmts}$	$[\![\underline{Stmts}_1]\!]_{Stmts}; \left(\mu X \bullet \left(\begin{array}{l} \textbf{if}\ [\![\underline{Expr}]\!]_{Expr} \longrightarrow \\ \quad [\![\underline{Stmts}]\!]_{Stmts}\ ;\ [\![\underline{Stmts}_2]\!]_{Stmts}\ ;\ X \\ [] \neg [\![\underline{Expr}]\!]_{Expr} \longrightarrow \textbf{skip} \\ \textbf{fi} \end{array} \right) \right)$

TABLE D.5. Supplementary translation rules for Java statements.

Rule	Application	Result
SR11	$[\![\underline{Obj}\texttt{.}\underline{Meth}\ \texttt{(}\underline{Args}\texttt{)}\ \texttt{;}\ ]\!]_{Stmts}$ For safelet and mission sequencer objects.	$[\![\underline{Meth}]\!]_{Name} Call\,!\,[\![\underline{Args}]\!]_{Expr} \longrightarrow$ $[\![\underline{Meth}]\!]_{Name} Ret \longrightarrow \textbf{skip}$
SR12	$[\![\underline{Var}\ \texttt{=}\ \underline{Obj}\texttt{.}\underline{Meth}\texttt{(}\underline{Args}\texttt{)}\ \texttt{;}\ ]\!]_{Stmts}$ For safelet and mission sequencer objects.	$[\![\underline{Meth}]\!]_{Name} Call\,!\,[\![\underline{Args}]\!]_{Expr} \longrightarrow$ $[\![\underline{Meth}]\!]_{Name} Ret\,?\,ret \longrightarrow [\![\underline{Var}]\!]_{Name} := ret$
SR13	$[\![\underline{Obj}\texttt{.}\underline{Meth}\ \texttt{(}\underline{Args}\texttt{)}\ \texttt{;}\ ]\!]_{Stmts}$ For mission and handler objects.	$[\![\underline{Meth}]\!]_{Name} Call\,.\,\mathsf{IdOf}(\mathsf{TypeOf}(\underline{Obj}))\,!\,[\![\underline{Args}]\!]_{Expr} \longrightarrow$ $[\![\underline{Meth}]\!]_{Name} Ret\,.\,\mathsf{IdOf}(\mathsf{TypeOf}(\underline{Obj})) \longrightarrow \textbf{skip}$
SR14	$[\![\underline{Var}\ \texttt{=}\ \underline{Obj}\texttt{.}\underline{Meth}\texttt{(}\underline{Args}\texttt{)}\ \texttt{;}\ ]\!]_{Stmts}$ For mission and handler objects.	$[\![\underline{Meth}]\!]_{Name} Call\,.\,\mathsf{IdOf}(\mathsf{TypeOf}(\underline{Obj}))\,!\,[\![\underline{Args}]\!]_{Expr} \longrightarrow$ $[\![\underline{Meth}]\!]_{Name} Ret\,.\,\mathsf{IdOf}(\mathsf{TypeOf}(\underline{Obj}))\,?\,ret \longrightarrow [\![\underline{Var}]\!]_{Name} := ret$

The function $TypeOf(\underline{Obj})$ determines the type of an object, and $\mathsf{IdOf}(type)$ the mission or handler identifier of a class type.

TABLE D.6. Translation rules for calls to interacting methods.

Rule	Application	Result
SR15	$[\![\underline{Var}\ \texttt{=}\ \texttt{new}\ \underline{Class}\texttt{(}\underline{Args}\texttt{)}\ \texttt{;}\ ]\!]_{Stmt}$ where $\underline{Class}$ is a safelet or mission sequencer class.	$[\![\underline{Var}]\!]_{Name} := \textbf{new}\ [\![\underline{Class}]\!]_{Name} Class([\![\underline{Args}]\!]_{Expr})$
SR16	$[\![\underline{Var}\ \texttt{=}\ \texttt{new}\ \underline{Class}\texttt{(}\underline{Args}\texttt{)}\ \texttt{;}\ ]\!]_{Stmts}$ where $\underline{Class}$ is a mission or handler class.	$[\![\underline{Var}]\!]_{Name} := \textbf{new}\ [\![\underline{Class}]\!]_{Name} Class([\![\underline{Args}]\!]_{Expr})$; $[\![\underline{Class}]\!]_{Name} Init\,.\,\mathsf{IdOf}(\underline{Class})\,!\,[\![\underline{Var}]\!]_{Name} \longrightarrow \textbf{skip}$

TABLE D.7. Translation rules for the creation of active objects.